MegaFox:

1002 Things You Wanted To Know About Extending Visual FoxPro®

Marcia Akins
Andy Kramek
Rick Schummer

Edited by
Steve Dingle

Hentzenwerke Publishing

MegaFox: 1002 Things You Wanted to Know About Extending Visual FoxPro

Marcia Akins
Andy Kramek
Rick Schummer

Hentzenwerke Publishing

Published by:
Hentzenwerke Publishing
980 East Circle Drive
Whitefish Bay WI 53217 USA

Hentzenwerke Publishing books are available through booksellers and directly from the publisher. Contact Hentzenwerke Publishing at:
414.332.9876
414.332.9463 (fax)
www.hentzenwerke.com
books@hentzenwerke.com

MegaFox: 1002 Things You Wanted to Know About Extending Visual FoxPro
By Marcia Akins, Andy Kramek, and Rick Schummer
Technical Editor: Steve Dingle
Copy Editor: Farion Grove

Copyright © 2002 by Marcia Akins, Andy Kramek, and Rick Schummer

All other products and services identified throughout this book are trademarks or registered trademarks of their respective companies. They are used throughout this book in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

All rights reserved. No part of this book, or the ebook files available by download from Hentzenwerke Publishing, may be reproduced or transmitted in any form or by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior written permission of the publisher, except that program listings and sample code files may be entered, stored and executed in a computer system.

The information and material contained in this book are provided “as is,” without warranty of any kind, express or implied, including without limitation any warranty concerning the accuracy, adequacy, or completeness of such information or material or the results to be obtained from using such information or material. Neither Hentzenwerke Publishing nor the authors or editors shall be responsible for any claims attributable to errors, omissions, or other inaccuracies in the information or material contained in this book. In no event shall Hentzenwerke Publishing or the authors or editors be liable for direct, indirect, special, incidental, or consequential damages arising out of the use of such information or material.

ISBN: 1-930919-27-1

Manufactured in the United States of America.

I want to dedicate this book to my father,
and I only wish that he was still alive to see its publication.
The only thing that would have made him prouder than KiloFox
would have been this, my second major published work.

—Marcia Akins

This work is dedicated to my mother,
who has always been there for me no matter what
I have done whether good or bad.
Anything that is good about what I am today
is due largely to her unfailing support and love
(the bad is entirely my own fault) and
I appreciate her more than mere words can express.

—Andy Kramek

This book is dedicated to my wife Therese,
who has supported and joined me on every adventure
I have dreamed up and decided to take. Without your love
and support over the past 20 years
I would not be the person that I am today.

—Rick Schummer

Our Contract with You, The Reader

In which we, the folks who make up Hentzenwerke Publishing, describe what you, the reader, can expect from this book and from us.

Hi there!

I've been writing professionally (in other words, eventually getting a paycheck for my scribbles) since 1974, and writing about software development since 1992. As an author, I've worked with a half-dozen different publishers and corresponded with thousands of readers over the years. As a software developer and all-around geek, I've also acquired a library of more than 100 computer and software-related books.

Thus, when I donned the publisher's cap five years ago to produce the *1997 Developer's Guide,* I had some pretty good ideas of what I liked (and didn't like) from publishers, what readers liked and didn't like, and what I, as a reader, liked and didn't like.

Now, with our new titles for 2002, we're entering our fifth season. (For those who are keeping track, the '97 DevGuide was our first, albeit abbreviated, season, the batch of six "Essentials" for Visual FoxPro 6.0 in 1999 was our second, and, in keeping with the sports analogy, the books we published in 2000 and 2001 comprised our third and fourth.)

John Wooden, the famed UCLA basketball coach, posited that teams aren't consistent; they're always getting better—or worse. We'd like to get better…

One of my goals for this season is to build a closer relationship with you, the reader. In order for us to do this, you've got to know what you should expect from us.

- You have the right to expect that your order will be processed quickly and correctly, and that your book will be delivered to you in new condition.
- You have the right to expect that the content of your book is technically accurate and up-to-date, that the explanations are clear, and that the layout is easy to read and follow without a lot of fluff or nonsense.
- You have the right to expect access to source code, errata, FAQs, and other information that's relevant to the book via our Web site.
- You have the right to expect an electronic version of your printed book to be available via our Web site.
- You have the right to expect that, if you report errors to us, your report will be responded to promptly, and that the appropriate notice will be included in the errata and/or FAQs for the book.

Naturally, there are some limits that we bump up against. There are humans involved, and they make mistakes. A book of 500 pages contains, on average, 150,000 words and several megabytes of source code. It's not possible to edit and re-edit multiple times to catch every last

misspelling and typo, nor is it possible to test the source code on every permutation of development environment and operating system—and still price the book affordably.

Once printed, bindings break, ink gets smeared, signatures get missed during binding. On the delivery side, Web sites go down, packages get lost in the mail.

Nonetheless, we'll make our best effort to correct these problems—once you let us know about them.

In return, when you have a question or run into a problem, we ask that you first consult the errata and/or FAQs for your book on our Web site. If you don't find the answer there, please e-mail us at books@hentzenwerke.com with as much information and detail as possible, including 1) the steps to reproduce the problem, 2) what happened, and 3) what you expected to happen, together with 4) any other relevant information.

I'd like to stress that we need you to communicate questions and problems clearly. For example…

- "Your downloads don't work" isn't enough information for us to help you. "I get a 404 error when I click on the **Download Source Code** link on www.hentzenwerke.com/book/downloads.html" is something we can help you with.
- "The code in Chapter 10 caused an error" again isn't enough information. "I performed the following steps to run the source code program DisplayTest.PRG in Chapter 10, and I received an error that said 'Variable m.liCounter not found'" is something we can help you with.

We'll do our best to get back to you within a couple of days, either with an answer or at least an acknowledgement that we've received your inquiry and that we're working on it.

On behalf of the authors, technical editors, copy editors, layout artists, graphical artists, indexers, and all the other folks who have worked to put this book in your hands, I'd like to thank you for purchasing this book, and I hope that it will prove to be a valuable addition to your technical library. Please let us know what you think about this book—we're looking forward to hearing from you.

As Groucho Marx once observed, "Outside of a dog, a book is a man's best friend. Inside of a dog, it's too dark to read."

Whil Hentzen
Hentzenwerke Publishing
October 2002

List of Chapters

Chapter 1: KiloFox Revisited 1
Chapter 2: Data Driving with VFP 25
Chapter 3: IntelliSense, Inside and Out 49
Chapter 4: Sending and Receiving E-mail 81
Chapter 5: Accessing the Internet 107
Chapter 6: Creating Charts and Graphs 137
Chapter 7: New and Improved Reporting 155
Chapter 8: Integrating PDF Technology 197
Chapter 9: Using ActiveX Controls 233
Chapter 10: Putting Windows to Work 303
Chapter 11: Deployment 325
Chapter 12: VFP Tool Extensions and Tips 367
Chapter 13: Working with Remote Data 415
Chapter 14: VFP and COM 471
Chapter 15: Designing for Extensibility 511
Chapter 16: VFP on the Web 543
Chapter 17: XML and ADO 577
Chapter 18: Testing and Debugging 629

Table of Contents

Our Contract with You, The Reader v
Acknowledgements xxi
About the Authors xxv
How to Download the Files xxvii

Chapter 1: KiloFox Revisited 1
- Updates to KiloFox 1
  - How do I clean up my working environment? 1
  - How do I convert character strings into data? 2
  - How do I determine whether a tag exists? 3
  - How do I use GOTO safely? 4
  - How do I extract a specified item from a list? 5
  - How can I browse field names when the table has captions? 6
  - How do I make a SQL generated cursor updateable? 6
  - How can I change the connection used by a Remote View? 7
  - How do I check my query's optimization? 8
  - How do I pop up a calendar from a grid cell? 9
  - How do I put a combo in a grid? 11
  - How do I run code when a projecthook is activated? 12
- Things that we missed in KiloFox 13
  - How do I set focus to a control? 13
  - How do I display the current record at the top of my grid? 16
  - How do I lock the leftmost column in my grid? 17
  - How do I create truly generic command buttons? 18
  - How do I set up a hot key to declare local variables? 20

Chapter 2: Data Driving with VFP 25
- What exactly is "data driving"? 25
  - The three different types of data 25
  - What goes into the metadata? 26
  - Where should metadata be stored? 27
- Why bother with data driving? 28
  - Performance overhead 28
  - Design considerations 28
  - Maintenance issues 29
  - So is data driving worth it? 29
- How do I data drive my menus? 29
  - What type of menus do we want to data drive? 30
  - MPR file structure for a shortcut menu 30
  - The shortcut menu metadata 31

The shortcut menu generator class 33
Using the shortcut menu class 35
How can I format text correctly? 35
The problem 36
The solution 36
The xchgcase class 37
How do I data drive object instantiation? 40
How do I data drive a migration? 43
How do I data drive data validation? 45

Chapter 3: IntelliSense, Inside and Out 49
IntelliSense in Visual FoxPro 49
What is IntelliSense? 49
How do I configure IntelliSense? 53
How do I work with the FoxCode table? 53
What are all these record types? 55
How do I create my own scripts? 58
How do I create a script to insert a block of code? 59
How do I create a script to generate a list? 61
How do I create my own Quick Info tips? 66
What is the Properties button in the IntelliSense Manager for? 67
How do I modify default behavior? 68
Putting IntelliSense to work 70
How do I change the behavior of browse? 70
How do I insert a header into a program? 71
How do I get a list of files? 72
How do I get a list of variables? 74
How do I get a list of all my custom shortcuts? 76
Isn't there an easier way to create a script? 78
Conclusion 79

Chapter 4: Sending and Receiving E-mail 81
What are the options? 81
What is all this alphabet soup, anyway? 81
How do I use MAPI? 83
How do I read mail using MAPI? 85
How do I send mail using MAPI? 90
What is CDO 2.0? 93
How do I send mail using CDO 2.0? 94
How does the *cusCDO* class work? 95
Can I control Outlook programmatically? 99
How do I access the address book? 100
How do I read mail using Outlook Automation? 101
How do I send mail using Outlook Automation? 104
Conclusion 104

Chapter 5: Accessing the Internet 107

How do I show a Web page in a form? 107
But when I run the form, I get an error! 107
Displaying content 108
How do I put a browser on the VFP desktop? 109
How do I print the contents of a Web page? 111
How do I extract data from a Web page? 112
Using the browser control's *ExecWB()* method 112
Using the document object's *ExecCommand()* method 113
Using the DOM 114
How do I create a hyperlink in a VFP form? 117
What about the FoxPro foundation classes? 118
Creating your own hyperlink classes 119
How do I use Web Services in my applications? 121
How do I register a Web Service using the VFP extensions? 122
How do I use a registered Web Service? 124
How do I find out how to use a Web Service? 128
The WSDL Inspector form 130
Conclusion 136

Chapter 6: Creating Charts and Graphs 137

Graphing terminology 137
How do I create a graph using MSChart? 139
How do I create a graph using MSGraph? 143
How do I create a graph using Excel Automation? 148
Conclusion 153

Chapter 7: New and Improved Reporting 155

Visual FoxPro Report Designer 155
What are the new features in the Visual FoxPro 7 Report Designer? 155
How do I prompt for a printer from preview mode? 157
How do I print watermarks on a report? 157
How do I disable the report toolbar printer button? 160
How do I detect if the user canceled printing and retain statistics for my reports? 162
Crystal Reports 165
Why should I consider Crystal Reports for reporting? 166
What techniques can be used to integrate Visual FoxPro data with Crystal Reports? 167
What do I need to set up to run the samples in this chapter? 168
What is the performance of the different techniques used to integrate Visual FoxPro data with Crystal Reports? 173
How do I create a report in Crystal Reports? 175

What happens when I change the structure of source cursor for the report? 179
How do I implement hyperlinks in a report? 180
How do I display messages from within a report? 181
How do I add document properties to a report? 182
How do I implement charts/graphs in a report? 183
How do I export reports to RTF, PDF, XML, and HTML formats? 184
How do I implement drill down in my reports? 185
How do I work with subreports in Crystal Reports? 187
What can I do with the Report Designer Component? 189
How do I work with the Crystal Report Viewer object? 190
What do I need to add to my deployment package when using Crystal Reports? 192
Crystal Report wrapper objects for commercial frameworks 194
What might you miss about the Visual FoxPro Report Designer when working with Crystal Reports? 194
Conclusion 195

Chapter 8: Integrating PDF Technology 197
Which version of Acrobat do I need? 197
What is needed to generate a PDF file? 198
How do I determine which PDF product to license? 199
How can I use PDF technology in my Visual FoxPro apps? 200
How do I output Visual FoxPro reports to PDF using Adobe Acrobat? 200
What are the errors to trap when printing to PDFs? 202
How do I run PDF reports unattended using Acrobat? 203
How do I run PDF reports unattended using Amyuni? 204
How do I email a Visual FoxPro report? 206
How can I replace the Visual FoxPro Report print preview? 209
How do I present Acrobat PDFs in a Visual FoxPro form? 212
What is Acrobat Forms Author technology? 220
How can I extract data out of a PDF form file? 224
Register the FDF Toolkit ActiveX control 224
Instantiating the object to access the FDF File 225
How do I prefill the PDF Form with data? 226
How can I merge PDF files together? 228
Conclusion 231

Chapter 9: Using ActiveX Controls 233
How do I include ActiveX controls in a VFP Application? 233
How do I find out what controls are in an OCX? 235
Okay, but how do I get the class name of an ActiveX control? 236
How do I add an ActiveX control to a form or class? 236
Putting ActiveX controls to use 237
How do I subclass an ActiveX control? 238

How do I use the Windows progress bar? 238
Setting up the progress bar class 239
Displaying the progress bar 240
How do I use the Date and Time Picker? 241
So what is the CheckBox property for? 243
How does the custom acxDTPicker class work? 244
How do I use the MonthView? 245
How do I use the ImageList? 246
How do I store images in the ImageList? 247
How do I bind the ImageList to other controls? 248
How do I use the ListView? 249
How do I add items to my ListView? 251
How do I sort the items in my ListView? 252
How do I know which item is selected? 253
Can I make the ListView behave like a data-bound control? 254
How do I use the ImageCombo? 259
How do I display a hierarchical list in the ImageCombo? 261
How do I use the TreeView? 263
How are Nodes added to the TreeView? 263
How do I navigate the TreeView? 264
How does the acxTreeView class work? 265
And finally 270
How do I synchronize a TreeView with a ListView? 270
Controls for animation and sound 272
How do I animate a form? 272
How do I add sound to my application? 274
How do I use other types of media in my application? 279
How do I add a status bar to a form? 282
Setting up a standard status bar 283
What's the point of the simple style status bar? 284
Managing the status bar dynamically 285
Conclusions about the status bar control 288
What is the Winsock control? 288
So which protocol is best? 288
How do I include messaging in my application? 289
How do I transmit error reports without using e-mail? 292
Winsock control—conclusion 301
ActiveX controls, the last word 301

Chapter 10: Putting Windows to Work 303
How do I work with the Windows Registry? 303
The structure of the Registry 303
So, when should I be using the Registry? 305
How do I access the Registry? 306
How do I read data from the Registry? 308

How do I write data to the Registry? 311
How do I change Visual FoxPro Registry settings? 313
Conclusion 314
What is the Windows Script Host? 314
Where can I get the Windows Script Host? 316
How do I use the Windows Script Host to automatically update my application? 318
How do I use the Windows Script Host to read the Registry? 320
How do I use the Windows Script Host to write to the Registry? 321
How do I let the user choose which printer to use? 322
How do I delete an entire folder? 323
How do I rename a directory? 323
How do I know whether a drive is ready? 323
Conclusion 324

Chapter 11: Deployment 325
How do I integrate graphic images into an EXE? 325
How do I create graphic images? 327
How do I deploy graphic images? 328
How do I get the version details from the executable? 328
Where should I install my application ActiveX controls? 330
Where do the Visual FoxPro runtimes have to be installed? 331
How do I know which runtime files are being used? 332
How can I distribute new versions of the runtime files? 332
How do I run a different Visual FoxPro runtime language resource? 333
What executable format can I release my application? 334
What installation scheme should I use? 335
File Server Install 336
Workstation Install 336
Data Install 337
Web Server Install 337
How do I package the install? 337
What are some handy utilities to ship? 338
Reindex and Database Updater 338
GenDBC/GenDBCX 339
Checking next id table 339
Configuration/control table updater 339
InstallShield Express for Visual FoxPro tips 340
Where do I find InstallShield Express? 340
What are the advantages of using InstallShield Express over the Setup Wizard? 341
What are the disadvantages of using InstallShield Express vs. Setup Wizard? 341
How do I upgrade to the full version of InstallShield Express? 342
How do I leverage the default Windows directories? 342

How do I work with setup types and features? 344
What is a merge module and which do I use for Visual FoxPro installs? 345
How do I create shortcuts or folders? 347
How do I create Registry keys? 348
How can I limit the hardware configurations the app will install? 349
How do I have the install files registered for all users of the computer? 349
Visual FoxPro 6 Setup Wizard tips 350
How do I run the Visual FoxPro 6 Setup Wizard? 350
How does the Setup Wizard retain its settings for the next build? 351
What tips are there for Step 1: Locate Files? 353
What tips are there for Step 2: Specify Components? 353
What tips are there for Step 3: Create Disk Image Directory? 353
What tips are there for Step 4: Specify Setup Options? 354
What tips are there for Step 5: Specify Default Destination? 355
What tips are there for Step 6: Change File Settings? 356
What tips are there for Step 7: Finish? 357
How do I AutoRun Visual FoxPro 6 installations? 358
What are the additional setup parameters? 359
How do I get a list of files and changes from the install? 360
How do I have a user reinstall an application? 360
How do I have a user uninstall an application? 360
How do I have a user install without intervention? 361
How can I create a desktop shortcut using the Setup Wizard? 361
How do I find out about Setup Wizard issues and bugs? 363
How can I ensure a smooth deployment? 363
Duplication 364
Users 364
Hardware 364
Training materials 364
Conclusion 365

Chapter 12: VFP Tool Extensions and Tips 367
Menus 367
How can I dynamically change captions in menu? 367
How can I permanently disable a menu option? 369
How can I dynamically disable menu bars in menu? 369
How can I remove menu pads and bars from a menu? 371
How can I create a menu to use as a template for my VFP apps? 372
How do I programmatically execute a VFP provided menu bar? 374
How do I include native VFP menu items in a custom menu? 374
How do I create and implement a shortcut menu? 375
How do I create and implement a top-level form menu? 377
How can I create a developer tool menu in VFP? 379
What happens if I need to compile a VFP 7 menu in VFP 6? 380
How can I fix the disabled menu after a report preview? 382

A partial replacement for the Menu Designer 382
Coverage Profiler 386
How do I start recording coverage logs? 386
What are the different columns in the coverage log files? 387
How do I register a Coverage Profiler add-in? 388
Where are Coverage Profiler preferences and add-in registrations saved? 388
How can I delete Coverage Profiler add-ins I no longer want registered? 389
Coverage Profiler add-in to summarize module performance 390
Class Browser 393
How can I set the default file to be opened when Class Browser is started? 393
How do I open the Class Browser with a specific class? 393
How can I move and copy classes between class libraries? 394
How do I rename methods and properties without opening the class? 394
How can I safely change a class name without breaking references to subclasses? 395
How can I test classes from the Class Browser? 395
How can I view and edit superclass code via the Class Browser? 397
Does the Class Browser add-in retain the Regional Settings for time and date? 397
How do I create a Class Browser add-in to set the font to my favorite? 399
Task List 401
How do I add my own custom fields to the Task List? 402
How can I use my custom fields in the Visual FoxPro Task List? 403
How can I add tasks programmatically to the Task List? 404
How can I update tasks programmatically in the Task List? 405
How can I delete tasks programmatically in the Task List? 406
How can I fix a Task List when it seems to have lost its mind? 407
What happens to the Task List tasks after I add an existing user-defined fields table? 408
Putting it all together with the G2 Task List Editor 408
Object Browser 409
How do I execute the Object Browser programmatically? 409
How do I get rid of “cached” objects? 410
How do I determine the values of constants defined in a COM object? 410
How can I use the Object Browser to create class templates to implement interfaces? 411
How do I find out the name of the OCX file to ship with my deployment setup? 412
Project Manager 412
How can I automate the author settings in the Project Info dialog? 412
Conclusion 413

Chapter 13: Working with Remote Data 415

Running the examples 415
Connecting to remote data 415
How do I connect to a database using ODBC? 415
How do I connect to a database using OLEDB? 417
Connecting to a database that is not installed locally 419
Which is better, ODBC or OLEDB? 419
How can I be sure users have the correct settings? 420
How do I use remote views in Visual FoxPro? 423
1. Configure the connection 423
2. Configure the remote data handling 426
3. Define a remote view 428
4. Create the form 431
Summary 432
What's wrong with remote views? 433
What should I use instead of remote views then? 434
FoxPro's SPT functions 435
Connection management 436
Command execution 438
Transaction management 440
Miscellaneous 442
Should I run in synchronous or asynchronous mode? 442
How do I work with SPT cursors? 443
How can I make a cursor updatable? 443
What are the data classes? 446
Defining cursors 451
How do I use the data classes? 456
Conclusion 470

Chapter 14: VFP and COM 471

What are COM and COM+? 471
So, COM is...? 471
How does it work? 471
And COM+ is... 472
Sounds cool, how could it be "legacy technology"? 473
All about interfaces 473
Late binding 473
Early binding 475
How does this apply to Visual FoxPro? 475
Working with COM in Visual FoxPro 480
What's the difference between single and multi-threaded DLLs? 480
Why are there two versions of the Visual FoxPro runtime library? 481
How does COM work? 482
What is "instancing"? 486
How do I create a COM DLL? 487

Designing COM components 490
How do I handle errors? 493
How do I implement an interface? 497
And there's more! 501
How can I use COM in the real world? 502
Building the component 502
Testing the component in Visual FoxPro 504
Testing the component with ASP 505
How do I distribute a component? 507
How do I register a component on my machine? 507
Conclusion 509

Chapter 15: Designing for Extensibility 511
How do I design an application? 511
Monolithic applications 511
Layered applications 512
So, I should design my application using layers then? 516
Layer pattern summary 516
Implementing design patterns in Visual FoxPro 517
What is a Bridge and how do I use it? 518
What is a Strategy and how do I use it? 522
What is a Chain of Responsibility and how do I use it? 526
What is a Mediator and how do I use it? 530
What is a Decorator and how do I use it? 535
What is an Adapter and how do I use it? 538
What is a wrapper, and how do I use it? 539
Conclusion 541

Chapter 16: VFP on the Web 543
How do I data drive the production of HTML? 543
How do I give my Web pages a consistent look and feel? 543
How do I generate HTML formatted lists? 547
Putting it all together 555
What are the Office Web Components? 557
How do I install the Office Web Components? 558
How do I create graphs using the Office Web Components? 558
How do I keep from having to take my Web server down when I modify my DLL? 564
How do I publish a Web Service? 566
What is a WSML file? 569
I don't expose my application on the Internet, so why should I bother with Web Services? 570
A sample Web Service 571
Conclusion 575

Chapter 17: XML and ADO 577

What is XML? 577
How does Visual FoxPro handle XML? 577
XML terminology 578
What parsers are available and which one should I use? 580
Does it matter which version of MSXML I use? 580
What are the most important properties and methods of the DOM? 581
How do I data-drive the production of XML? 584
How do I data drive importing XML into cursors? 591
How do I use the SAX interface to import XML? 591
How do I use the DOM to import XML? 596
How do I validate an XML document using a schema? 598
How do I create an XDR schema? 598
How do I create an XSD schema? 600
How do I use the SchemaCache to validate XML documents? 605
What is XSLT? 606
What are XSL patterns? 607
What XSLT elements do I use to define my template? 608
How do I use XSLT to transform my XML documents? 609
How do I use the DOM's XSL processor to transform XML? 613
Conclusion 614
What is ADO? 614
The ADO object model 615
How do I convert a cursor into an ADO RecordSet? 625
How do I convert an ADO RecordSet into a cursor? 627
Conclusion 628

Chapter 18: Testing and Debugging 629

How do I know an application is ready? 630
What types of testing can be performed? 631
Unit testing 632
Integration testing 633
System testing 635
User acceptance testing 637
Regression testing 637
What is a test plan? 638
How do I test various types of releases? 640
How do I manage the risk of releasing defects? 641
How can I test forms? 641
How can I test reports and labels? 643
How can I test business objects? 645
How can I test other components? 648
How do I test systems to verify source code is not in the path? 648
How do I avoid "Feature not available" errors? 649
What are walkthroughs and what are the benefits? 649

What different types of walkthroughs can you do? 651
How does a developer prepare for a walkthrough? 652
What is the reviewer's responsibility of a walkthrough? 654
What happens during the walkthrough? 655
What are the outcomes of a walkthrough? 656
What is an alternative to performing a walkthrough? 657
Why should I consider hiring someone to test? 657
Developers are the worst testers of their own code 657
Customers are not good at testing applications 657
You will be a better developer 658
Avoid the trap that you cannot afford to hire testers 658
How can I use the Coverage Profiler to test code? 659
What types of automatic test tools are available? 661
How can I log defect reports? 662
So what kind of information should be tracked on reported defects? 662
What mechanisms are available to track defects? 663
How can I test apps on various platforms without reloading the OS? 664
Debugging is different from testing 666
What is the scientific method approach to debugging? 666
Make an observation 666
Formulate questions 667
Create hypothesis/prediction 667
Fix and test 667
Evaluate results 668
Decision 668
Visual FoxPro debugger tips 668
How can I set the debugger configuration to factory settings? 669
How can I save and restore the configuration of the debugger? 671
How can I reorder the contents of the watch window without deleting and re-entering each expression? 672
How can I track which events were triggered in my code? 672
How can I track which methods were executed in my code? 673
How can I change values of memory variables in the debugger? 674
How can I ensure variables are declared? 674
What are some general tips and tricks for the debugger? 676
How can I get quick access to the property values of a specific object? 677
Conclusion 677

Acknowledgements

We said in the acknowledgements to *1001 Things You Wanted to Know About Visual FoxPro* that it was impossible to acknowledge individually all those who contributed to the book. That is even more true for this book, which severely tested our individual abilities and made us even more reliant on the support and assistance of the FoxPro community that we all feel so lucky to be a part of. Indeed it was largely the positive response of the FoxPro community to *1001 Things* that prompted us all to get back together and tackle this project.

First we have to thank our Technical Editor, Steven Dingle. The work of the Technical Editor is not as glamorous as that of being an author, and people tend to forget that the Technical Editor is not only an integral part of the team but is largely responsible for the final shape and structure of the book. It is a difficult and often thankless task. Steve is without doubt the best Technical Editor we have been lucky enough to work with. Not only does he have great VFP skills in his own right, he also has more of a knack for asking the most awkward, difficult, and penetrating questions than anyone we know. Without his persistence and insight this book would have looked very different and would, we are sure, not have been as good as it is. We all owe him a large debt of thanks.

Next we must thank Whil Hentzen, first for allowing us to write a second book and second for continuing to support the FoxPro community so prolifically—the first FoxPro Lifetime Achievement Award could not have been bestowed on a better person. Of course, we know that Whil does not actually do it *all* himself, and our thanks go out also to the team at Hentzenwerke who turn our raw text into polished and professional looking work.

Every author always thanks the community as a whole, but in our case we really could not have written the book without the community. Unfortunately, we cannot hope to name everyone individually whose question, comment, or answer has prompted something in this book. The various forums (FoxForum.com, CompuServe, Virtual FoxPro User Group, Fox Wiki, West Wind Threads, and UniversalThread) have provided the inspiration for most of the things included in this book. Having said that, there are some individuals whose contributions to this book have been very specific and they deserve a special mention:

- Steven Black, who not only coined the nickname *KiloFox* for *1001 Things* but who also designed and implemented the original Lister and Render classes on which those presented in Chapter 16 are based.
- George Tasker, for his assistance and suggestions when reviewing Chapter 10.
- Erik Moore, without whose expertise Chapter 17 would have been very different.
- Toni Feltman, for her help in getting our heads around XSLT in Chapter 17.
- Walter Meester, for sharing his tip on making the current row the top one in a grid in Chapter 1.
- Ed Leafe, for his help with Web Services and for allowing us to use his site for testing the code presented in Chapters 5 and 16 (did you know we did that, Ed?).

- Trevor Hancock, for sharing his utility to automatically extract text into an IntelliSense script in Chapter 3.
- Rick Strahl, for all his free classes and white papers, which we used extensively as reference material.
- Pamela Thalacker, for alpha/beta testing the G2 Task List Manager in Chapter 12.
- Paul Mrozowski, for his insight with Acrobat (Chapter 8) and Crystal Reports (Chapter 7).

Finally, we have to thank you, the reader, for two reasons. First, because just as *MegaFox* was being wrapped up for publication, we had to stop and re-visit *KiloFox* because it had sold out and needed a second edition. That was entirely unexpected when were writing the book and very gratifying, and we thank you for it. Second, for buying this book. We hope that it will live up to your expectations, and we have all tried very hard to meet the standards that you, rightly, expect and demand.

—Marcia Akins
—Andy Kramek
—Rick Schummer

August 2002

Afterword from Rick

Thanks to Whil, Marcia, and Andy for asking me back for round two. It was a pleasure collaborating on another book. Thanks to you guys, I now have a better (but not complete) understanding of how mothers block out the memories of childbirth and continue having more children, although I think the readers of *KiloFox* were the single biggest reason I decided to take part in *MegaFox*. Their encouraging feedback to *KiloFox* was so overwhelming that I could not have turned the opportunity down even if I wanted to.

I want to especially thank my family for their support while I immersed myself into the process of writing my chapters. To my wife and best friend Therese, who has almost endless patience, and gives the best back rubs, I love you more and more each day. To Chris, Nicole, and Amanda, our children who are still on this earth, thanks for making this process go faster by continuously asking when I would be done writing. You three make me the proudest dad on the planet. Our angel Paul is my constant reminder of what is important in life; please keep watching over us from afar. A special thanks to my parents (all four of them) for instilling strong values, the importance of dedication to improving yourself, and for expecting nothing but my best in everything I do.

Steve Bodnar and Steve Sawyer, my friends and business partners at Geeks and Gurus, Inc., have been very supportive of the time this book took away from my dedication to growing our new company that much more. Thanks for reviewing some concepts included in this book, providing me the important feedback, and making going to work fun again.

I also want to thank Patty Nowak, who will always be my first editor. Thanks for being my friend, telling me what I do right, what needs to be corrected, and for challenging my thinking. I am a better writer because of your insight, and a better developer because of the standards you have held me to.

Finally, to the bottlers of Coca-Cola, your fine product makes it easier to write code and books at all hours, day and night.

—Rick Schummer

—August 2002

About the Authors

Marcia Akins

Marcia (with husband, Andy) is joint owner of Tightline Computers, Inc. The company is located in Akron, Ohio, and specializes in the provision of expert assistance and support in all phases of the Software Development life cycle. Marcia has been the recipient of the Microsoft Most Valuable Professional award since 1999 and also has Microsoft Certified Professional qualifications for both Distributed and Desktop Applications in Visual FoxPro.

Marcia's published work includes articles for both *FoxPro Advisor* (Advisor Publications) and *FoxTalk* (Pinnacle Publishing) and the very successful book *1001 Things You Wanted to Know About Visual FoxPro* (Hentzenwerke Publishing, May 2000). She has also been writing the column "The Kit Box" in *FoxTalk* with her husband and colleague, Andy Kramek, since December 2001. Speaking engagements include Great Lakes Great Database Workshop (Milwaukee), Advisor DevCon (San Diego and Fort Lauderdale), EssentialFox (Kansas City), Conference to the Max (Holland), Praha DevCon (Czech Republic), and European DevCon (Frankfurt), as well as user group meetings in Europe and the US.

When she is not busy developing software, Marcia enjoys spending time on the golf course and in the gym. She also enjoys travel and reading Nero Wolfe novels. Finally, she is still very happy when she is harassing Andy.

You can reach Marcia at marcia@tightlinecomputers.com.

Andy Kramek

Andy (with wife, Marcia) is joint owner of Tightline Computers, Inc. The company is located in Akron, Ohio, and specializes in the provision of expert assistance and support in all phases of the Software Development life cycle. As well as being a Microsoft Most Valuable Professional he is also a Microsoft Certified Professional for Visual FoxPro in both Desktop and Distributed applications. He has been active for many years on the FoxPro support forums on CompuServe, where he is also a SysOp, and the Virtual FoxPro Users Group. He has spoken at user groups and conferences all over the world, including Advisor DevCons in San Diego and Fort Lauderdale, at GLGDW (Milwaukee), and European DevCons in Frankfurt, the UK, the Netherlands, and Czech Republic.

Andy's other published work includes *The Revolutionary Guide to Visual FoxPro OOP* (Wrox Press, 1996), and, together with Marcia Akins and Rick Schummer, the 2001 UT Members Choice Book of the Year *1001 Things You Wanted to Know About Visual FoxPro*, more widely known as *KiloFox*. For more than four years he has written the monthly column "The Kit Box" in *FoxTalk* (Pinnacle Publishing), for the first three years with his friend and colleague Paul Maskens, and latterly with his wife, Marcia Akins.

In his free time Andy is a keen golfer and voracious reader.

You can reach Andy at andykr@tightlinecomputers.com.

Rick Schummer

Rick is a partner at Geeks and Gurus, Inc., in Detroit, Michigan. Geeks and Gurus aims to be a one-stop shop for small to medium size organizations that need help with databases, custom software, networks, Visual FoxPro mentoring, and Web-related services. He enjoys working with top-notch developers, and he has a passion for developing software using best practices, and for surpassing customer expectations, not just meeting them. After hours he writes developer tools that improve productivity and occasionally pens articles for *FoxTalk* (Pinnacle Publishing), *FoxPro Advisor* (Advisor Publications), and several user group newsletters.

Rick is a Microsoft Most Valuable Professional (VFP) and a Microsoft Certified Professional. He is founding member and Secretary of the Detroit Area Fox User Group (DAFUG), and he presents at user groups across North America, and at GLGDW 2000-2002, EssentialFox 2002, and VFE DevCon 2K2 conferences.

He spends his free time with his family, cheers the kids as they play soccer, has a volunteer role with the Boy Scouts, and loves spending time camping, cycling, coin collecting, and reading, and recently completed the Sterling Heights Citizen Fire Academy.

You can reach Rick at raschummer@geeksandgurus.com.

Steve Dingle

Steve is an independent consultant with more than 10 years of experience in designing and developing data-related applications. At the time of this writing he is migrating from North Carolina, USA to London, England to start a consulting company. He is also a Microsoft Certified Professional for Visual FoxPro in Desktop applications and has been an MVP (Microsoft Most Valuable Professional) since 1996. He can often be found on the FoxPro support forum on CompuServe, where he is also a SysOp.

Steve's ramblings have been published both in *FoxPro Advisor* (Advisor Publications) and *FoxTalk* (Pinnacle Publishing). He was also the Tech Editor for *Effectve Techniques for Application Development with VFP 6.0* (Hentzenwerke Publishing).

As for free time, Steve's two main passions, outside of writing code, are travel and playing chess.

You can reach Steve at Steve@SteveDingle.com.

How to Download the Files

Hentzenwerke Publishing generally provides two sets of files to accompany its books. The first is the source code referenced throughout the text. Note that some books do not have source code; in those cases, a placeholder file is provided in lieu of the source code in order to alert you of the fact. The second is the e-book version (or versions) of the book. Depending on the book, we provide e-books in either the compiled HTML Help (.CHM) format, Adobe Acrobat (.PDF) format, or both. Here's how to get them.

Both the source code and e-book file(s) are available for download from the Hentzenwerke Web site. In order to obtain them, follow these instructions:

1. Point your Web browser to www.hentzenwerke.com.
2. Look for the link that says "Download"
3. A page describing the download process will appear. This page has two sections:

- **Section 1:** If you were issued a username/password directly from Hentzenwerke Publishing, you can enter them into this page.
- **Section 2:** If you did not receive a username/password from Hentzenwerke Publishing, don't worry! Just enter your e-mail alias and look for the question about your book. Note that you'll need your physical book when you answer the question.

4. A page that lists the hyperlinks for the appropriate downloads will appear.

Note that the e-book file(s) are covered by the same copyright laws as the printed book. Reproduction and/or distribution of these files is against the law.

If you have questions or problems, the fastest way to get a response is to e-mail us at books@hentzenwerke.com.

Icons used in this book

Indicates that the referenced material is available for download at www.hentzenwerke.com.

Indicates information of special interest, related topics, or important notes.

Indicates a tip, trick, or workaround.

Indicates a warning or “gotcha.”

Indicates version issues.

Chapter 1
KiloFox Revisited

Although this book is not a sequel to *1001 Things You Wanted to Know About Visual FoxPro*, we did feel it was incumbent upon us to update some things from that book. Either we have found another way of doing something, or Visual FoxPro itself has changed and made things easier. There are also some new things that people have asked us about or that we have found ourselves since we finished work on KiloFox nearly two years ago.

Updates to KiloFox

Some of the solutions that we presented in KiloFox have either been amended, or even superceded, by changes in the functionality of the latest version of Visual FoxPro. We would be remiss in our responsibility to you, our reader, if we did not address these issues before going on to share our new tips. So, without further ado, here are some changes to the original tips presented in KiloFox.

How do I clean up my working environment? (*Example: ClearAll.prg*)

This is always a very personal issue. All developers have their own preferences for setting up their development environment and we are no exception to that rule. As described in KiloFox (page 25), we always use a little program to handle this chore. However, one of the problems with cleaning up the environment involves handling leftover data sessions. The code presented in KiloFox attempted to address this issue by using the *_Screen.Forms* collection to find open data sessions and close them down safely. However, it was never an entirely satisfactory solution because it did not address the issue of data sessions that were not associated with forms.

This became a critical matter when the *Session* base class was introduced in Service Pack 3 for Visual FoxPro 6.0. This class gave us the ability to create a data session that was totally divorced from any form and proved extremely useful for handling functionality that requires data, but which is not associated with a particular form. Examples include classes that deal with error or message handling and that use tables to store the associated text.

By basing such classes on the *Session* class, we can keep their tables separate from the general application environment in a private datasession of their very own. The consequence of adopting this approach is that we really do have to find a way of locating and closing these data sessions. Hitherto the only real solution involved looping through all 65,000 possible data sessions and trying to switch to each while error handling is disabled, like this:

```
LOCAL lnCnt
ON ERROR *
FOR lnCnt = 1 TO 65000
  SET DATASESSION TO (lnCnt)
NEXT
ON ERROR
```

The problem with this approach is that even Visual FoxPro takes several seconds to do it. Unfortunately there is no way to avoid processing all possible sessions because Visual FoxPro never renumbers them once they have been created. That means that it is entirely possible to have gaps in the sequence of data session numbers so you cannot simply exit from the loop as soon as the first invalid session ID is encountered.

Among the new functions introduced in Visual FoxPro 7.0 is **ASESSIONS()**, which specifically addresses this issue and which builds an array of all active data sessions. Note that **ASESSIONS()** does not tell us anything about the data sessions it finds (like which object owns it, or whether it actually contains anything) but merely logs their ID Numbers. So we can now write code that will check *all* data sessions for pending transactions and changes. Here is the relevant part of the revised CLEARALL.PRG (which can be found in the download files for this chapter).

```
*******************************
*** Revert Tables and Close Them
*******************************
*** Get the list of sessions
lnSess = ASESSIONS( laSess )
FOR lnCnt = 1 TO lnSess
  SET DATASESSION TO ( laSess[lnCnt] )
  *** Roll Back Any Transactions
  IF TXNLEVEL() > 0
    DO WHILE TXNLEVEL() > 0
      ROLLBACK
    ENDDO
  ENDIF
  lnCntUsed = AUSED( laUsed )
  *** Revert any pending changes too!
  FOR lnSessCnt = 1 TO lnCntUsed
    SELECT ( laUsed[lnSessCnt,2] )
    IF CURSORGETPROP( 'Buffering' ) > 1
      TABLEREVERT( .T. )
    ENDIF
    USE
  NEXT
NEXT
```

How do I convert character strings into data? (*Example: Str2Exp.prg*)

This is a problem that has always been around in Visual FoxPro because Combo and List boxes store the elements in their internal lists as string values. Setting the control's *BoundTo* property to true allows you to bind to a numeric *ControlSource*, but this affects only the data type of the *value* and not the data types of the list items. So whenever you need to use these items to update, or seek in the original data, you first have to convert back into the appropriate data type. In KiloFox (page 43) we introduced the *Str2Exp()* function to handle this issue. This function was designed to handle the type of character data used in Combos and Lists or generated by the enhanced VFP **TRANSFORM()** function.

However, the rapid growth in developing applications to run on the World Wide Web has forced us all to re-examine the way in which we handle and process character string data (which is, after all, at the root of both HTML and XML). Indeed, many of the language enhancements introduced in Visual FoxPro 7.0 are improvements in the handling and

manipulation of character data for precisely this reason. In this context, the original *Str2Exp()* function was very limited in its ability to deal with Date and DateTime values because it required that dates be in a format recognizable by the native Visual FoxPro **CTOD()** or **CTOT()** functions.

The revised version of this function specifically addresses this issue by providing a much more sophisticated treatment of Date and DateTime values. In fact, while superficially very similar to the original, this version offers several other minor enhancements over the original as a result of expanding its intent to handle data that originated, or was intended for display, in a Web browser. The new STRTOEXP.PRG can be found in the download files for this chapter.

How do I determine whether a tag exists? (*Example: IsTag.prg*)

This is another one of the "useful functions" from KiloFox (page 44) that can be simplified by taking advantage of new functionality in Visual FoxPro 7.0. It was intended to accept the name of an index tag and, optionally, a table alias and return a logical value indicating whether that tag name was defined for the table. The new **ATAGINFO()** function creates an array with significantly more information about the indexes defined for a table, as **Table 1** shows.

***Table 1**. ATagInfo() array definition*

Column	Content
1	Tag name (.idx name, if open)
2	Tag type (Primary, Candidate, Unique, or Regular)
3	Key expression
4	Filter
5	Order (Ascending or Descending)
6	Collate sequence

Here is the revised function, which also makes use of enhancements to the **ASCAN()** function:

```
*********************************************************************
* Program....: ISTAG.PRG
* Compiler...: Visual FoxPro 07.00.0000.9262 for Windows
* Abstract...: Passed the name of an index tag returns true if it is a
* ...........: tag for the specified table. Uses table in the current
* ...........: work area if no table name is passed.
*********************************************************************
FUNCTION IsTag( tcTagName, tcTable )
LOCAL ARRAY laTags[1]

*** Did we get a tag name?
IF TYPE( 'tcTagName' ) # 'C'
  *** Error - must pass a Tag Name
  ERROR '9000: Must Pass a Tag Name when calling ISTAG()'
  RETURN .F.
ENDIF
*** How about a table alias?
IF TYPE('tcTable') = 'C' AND ! EMPTY( tcTable )
  *** Get all open indexes for the specified table
  ATagInfo( laTags, "", tcTable )
```

```
ELSE
  *** Get all open indexes for the current table
  ATagInfo( laTags, "" )
ENDIF
*** Do a Case Insensitive, Exact=ON, Scan of the 1st column
*** Return Whether the Tag is Found or not
RETURN (ASCAN( laTags, tcTagName, -1, -1, 1, 7) > 0)
```

How do I use GOTO safely?

As was pointed out in KiloFox (page 52), the main problem with the **GOTO** command is that it performs no boundary checking with the result that if you try and go to a record number that is outside the range of currently valid records, it generates an error. As a solution we offered the *GoSafe()* function, which wrapped the attempt to move the record pointer inside an error trap. An alternative approach, suggested by several people, is to use the **LOCATE** function to position the record pointer, like this:

```
LOCATE FOR RECNO() = <nRecNum>
IF EOF()
  *** Record number is not valid
ENDIF
```

The rationale for this is that if **LOCATE** fails, it merely positions the record pointer at end-of-file and does not generate an error. This is, indeed simpler than the original function we devised but it does have one minor disadvantage. Even in VFP 7.0, the **LOCATE** command is still scoped to the current work area and cannot accept an "**IN <alias>**" clause. This does mean that you have to handle the issues associated with changing work area in your code, but that is a minor issue. Here is an alternative function based on this approach (downloadable as GOTOREC.PRG).

```
*********************************************************************
* Program....: GOTOREC.PRG
* Purpose....: Go to specified record - safely. Returns Record number
* ...........: or 0 if fails
*********************************************************************
FUNCTION GoToRec( tnRecordNumber, tcAlias )
LOCAL lnRetVal, lnSelect, lcAlias, lnRec
lnRetVal = 0
lnSelect = SELECT()
*************************************************************
*** Default Alias to currently selected if not passed
*************************************************************
lcAlias = IIF( VARTYPE(tcAlias) = "C" AND NOT EMPTY(tcAlias),;
               UPPER(ALLTRIM(tcAlias)), ALIAS() )
lnRec = IIF( VARTYPE(tnRecordNumber) = "N" AND NOT EMPTY(tnRecordNumber),;
             tnRecordNumber, 0 )
IF EMPTY( lnRec) OR EMPTY( lcAlias )
  *** Either no record number was passed or
  *** we cannot determine what table to use
  RETURN lnRetVal
ENDIF
```

```
*************************************************************
*** And select required alias
*************************************************************
IF NOT ALIAS() == lcAlias
  IF USED( lcAlias )
    SELECT (lcAlias)
  ELSE
    *** Specified Alias is not in use
    RETURN lnRetVal
  ENDIF
ENDIF
*************************************************************
*** Now do the LOCATE
*************************************************************
LOCATE FOR RECNO() = lnRec
lnRetVal = IIF( FOUND(), lnRec, 0 )
*************************************************************
*** Tidy Up and Return
*************************************************************
SELECT (lnSelect)
RETURN lnRetVal
```

How do I extract a specified item from a list?

In KiloFox (page 49) we presented a function named *GetItem()* that returned the specified entry from a delimited list and allowed you to specify the delimiter to use. The functionality could have been obtained by using two functions, **WORDS()** and **WORDNUM()**, which have long been in the FoxTools library. However, since it was not always possible to rely on having this library available we chose to develop *GetItem()* using only native commands and functions.

Version 7.0 has added two new functions, **GETWORDCOUNT()** and **GETWORDNUM()**, which replicate the functionality of their FoxTools equivalents and which can now be safely used to streamline the *GetItem()* function. We still need this function; not only because we don't want to have to re-visit and change all our existing code, but also because it does things slightly differently from the native functions.

- *GetItem()* returned a **NULL** when the end of the string was reached, or when the item index exceeded the number of items in the string. Under the same conditions, **GETWORDNUM()** returns only an empty string.
- *GetItem()* assumed a comma as the default separator if nothing was passed, while **GETWORDCOUNT()** and **GETWORNUM()** both allow any of Space, Tab, or Carriage Return characters as default delimiters.
- *GetItem()* defaults to the first entry if the index is omitted or is non-numeric, while **GETWORNUM()** generates either a "Too few arguments" or a "Function, argument, value type, or count is invalid" error under the same conditions.

Here is the revised function:

```
**********************************************************************
* Program....: GETITEM.PRG
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Extracts the specified element from a list
**********************************************************************
FUNCTION GetItem( tcList, tnItem, tcSepBy )
LOCAL lcRetVal, lcSepBy
lcRetVal = ""
*** Default to Comma Separator if none specified
lcSep = IIF( VARTYPE(tcSepBy) # 'C' OR EMPTY( tcSepBy ), ',', tcSepBy )
*** Default to First Item if nothing specified
tnItem = IIF( TYPE( 'tnItem' ) # "N" OR EMPTY( tnItem ), 1, tnItem)
*** If we have exceeded the length of the string, return NULL
IF tnItem > GETWORDCOUNT( tcList, lcSep )
  lcRetVal = NULL
ELSE
  *** Get the specified item
  lcRetVal = GETWORDNUM( tcList, tnItem, lcSep )
ENDIF
*** Return result
RETURN ALLTRIM(lcRetVal)
```

How can I browse field names when the table has captions?

This was always a problem prior to Version 7.0, and in KiloFox we presented a little wrapper program (page 77) that substituted the field name for the caption when browsing a table. Once again Visual FoxPro 7.0 has come to our rescue and the only change in the new version to the **BROWSE** command is the addition of a **NOCAPTIONS** clause to suppress the default display of captions, as illustrated in **Figure 1**.

The "*clients*" table has been opened with a **BROWSE NOCAPTIONS** command and, as you can see, we get the actual field names. The "*clients_a*" table used a plain **BROWSE** and so displays the captions for the fields, not the field names.

Clients

Iclientpk	Cclientid	Ccompany	Caddress	Ccity	Cregion	Czip
221	GOURL	Gourmet Lanchonetes	Av. Brasil, 442	Campinas	SP	04876-786
222	GREAL	Great Lakes Food Market	2732 Baker Blvd.	Eugene	OR	97403
223	GROSR	GROSELLA-Restaurante	5ª Ave. Los Palos Grandes	Caracas	DF	1081

Clients_a

Iclientpk	Client ID	Company Name	Address	City	State/Province	Postal Code
221	GOURL	Gourmet Lanchonetes	Av. Brasil, 442	Campinas	SP	04876-786
222	GREAL	Great Lakes Food Market	2732 Baker Blvd.	Eugene	OR	97403
223	GROSR	GROSELLA-Restaurante	5ª Ave. Los Palos Grandes	Caracas	DF	1081

***Figure 1**. Browse NoCaptions in action.*

How do I make a SQL generated cursor updateable?

In Visual FoxPro using the **CREATE CURSOR** command has always created an updateable cursor, but the cursor generated as the result of a SQL query against local Visual FoxPro tables has always been Read-Only. Until the advent of Version 7.0 the simplest way to make such a cursor updateable was to use a trick that forced Visual FoxPro to create a new (updateable) cursor with the same structure as the one generated by the SQL statement, thus:

```
SELECT * FROM clients INTO CURSOR junk NOFILTER
USE DBF( 'junk' ) AGAIN IN 0 ALIAS UpdCursor
```

This worked, and continues to work, perfectly well in all versions of Visual FoxPro *providing* that you ensure that Visual FoxPro does not simply create a filtered view of the underlying table (the introduction of the **NOFILTER** clause in Version 5.0 removed the necessity to include an explicit '**WHERE .T.**' on such queries to avoid getting a filtered view). However, there is one potential problem with this technique. It leaves the intermediate "junk" cursor in your data session and, unless you explicitly remove it, you will later have problems if you try to reuse the same temporary cursor name.

Fortunately, all this has changed in Visual FoxPro 7.0 and we can now consign this tip to history. The **SQL SELECT** statement now supports an additional **READWRITE** clause that will create an updateable cursor directly. Thus, the preceding code can be replaced by:

```
SELECT * FROM clients INTO CURSOR updCursor READWRITE
```

Of course, all that this does is to simplify the task of making the cursor updateable. There is still no direct mechanism in Visual FoxPro to send changes made to such a cursor back to the source tables; that functionality remains specific to Local Views.

How can I change the connection used by a Remote View?

There are many scenarios in which it would be useful to be able to redefine, at run time, the connection over which a view would retrieve its data. Perhaps the most obvious one is where there are several "versions" of the same database and either different users need to connect to different versions (very common in accounting systems), or the same user needs to connect to different places at different times (maybe to either work with *Test* or *Production* data). However, in all versions of Visual FoxPro prior to Version 7.0, a Remote View must use the same Visual FoxPro connection object that was used when the view was created. This means that the only way to change the data source to which the view connects is to redefine the view. Not only is this inflexible, this inability to dynamically determine the connection has been a major limitation of the implementation of Remote Views in Visual FoxPro.

A particularly welcome change, introduced in Version 7.0, is the easing of this particular restriction. The word "easing" is used advisedly because you can only override the old behavior if you open your view explicitly, using the new **CONNSTRING** clause, with the **USE** command. If you use the Form's DataEnvironment your views are still forced to use their default connection.

The **CONNSTRING** clause, as implied by its name, allows for an ODBC connection string to be specified as the view is being opened. The following code defines a Remote View using a locally defined connection object named *ConBase*:

```
CREATE SQL VIEW "RV_USADDR";
  REMOTE CONNECT "ConBase" AS ;
  SELECT AD.cadd1, AD.cadd2, AD.caddcity, Address.caddprovst, AD.caddpcode ;
    FROM dbo.address AD ;
   WHERE AD.caddcntry = 'USA'
```

The standard behavior, in all versions of Visual FoxPro, is to open the view using the same connection that was used when it was created (in this case, *ConBase*). All that is needed is the normal **USE** command:

```
USE rv_usaddr
```

Beginning with Version 7.0 you can now specify that the view connect explicitly to a different database by supplying the necessary connection string as well. Notice that you can use either a pre-defined DSN, or specify a database explicitly in the connection string. Either will work—as illustrated here:

```
lcConStr = "DRIVER=SQL Server;SERVER=(local);UID=xx;PWD=yy;DATABASE=Test"
USE rv_usaddr CONNSTRING ( lcConStr )
lcConStr = "UID=xx;PWD=yy;DSN=TestData"
USE rv_usaddr CONNSTRING ( lcConStr )
```

There is a marginal advantage to using the DSN in that it does hide some of the complexity of the connection parameters and, more importantly, ensures that if parameters change, only the DSN definition needs to be changed. Notice that the same result can be achieved interactively by passing an empty string instead of the connection detail. This forces the "Select Data Source" dialog to be displayed:

```
lcConStr = ""
USE rv_usaddr CONNSTRING ( lcConStr )
```

How do I check my query's optimization? (*Example: SQLOpt.prg*)

Another welcome change in Version 7.0 is the addition to the **SYS(3054)** function of an option to write the optimization details to a memory variable. This removes the necessity of manipulating the settings for **CONSOLE** and **ALTERNATE** that have been, hitherto, the only practical way of recording the results of the optimization check. A nice enhancement is the option to include the actual SQL statement as part of the display, which means that it is much easier to identify which results belong to which query. The following little program demonstrates how the new functionality can be used:

```
**********************************************************************
* Program....: SQLOPT.PRG
* Compiler...: Visual FoxPro 07.00.0000.9262 for Windows
* Purpose....: Illustrate the changes to SQL ShowPlan reporting
**********************************************************************
LOCAL lcOpt
*** Set up optimization (both Join and Filter) reporting
*** Include SQL statement and direct output to local variable
SYS(3054, 12, "lcOpt" )

*** Run the first query
SELECT CL.cclientid, CL.ccompany, CO.cfirst, CO.clast, PH.cnumber, LD.clddesc ;
  FROM  clients CL, contacts CO, phones PH, ludetail LD ;
 WHERE CL.iclientpk = CO.iclientfk ;
   AND CO.icontactpk = PH.icontactfk ;
   AND PH.iphonetypefk = LD.ildpk ;
```

```
   AND CL.ccountry = "USA" ;
  INTO CURSOR junk
*** Transfer contents of variable to file
STRTOFILE( lcOpt + CHR(13) + CHR(10), 'ChkOpt.txt' )

*** And then the second
SELECT * FROM clients WHERE ccountry = "USA" INTO CURSOR junk
*** Transfer contents from variable to file
STRTOFILE( lcOpt, 'ChkOpt.txt', .T. )
*** Turn off reporting, tidy up and review results
SYS( 3054, 0 )
CLOSE TABLES ALL
MODIFY FILE chkopt.txt NOWAIT
```

How do I pop up a calendar from a grid cell? (*Example: CH01.vcx::AcxCalendar and CalendarDemo.scx*)

In Chapter 4 of KiloFox (page 118), we presented a composite class that was designed to mimic the behavior of a drop-down list for entering dates. One of the shortcomings of this "calendar combo" was that it could not be used inside of a grid, so we decided to write a pop-up calendar form that could be used anywhere. Since we already had a custom date text box class (KiloFox page 88), we modified it to pop up the form calendar when the user double-clicked on it.

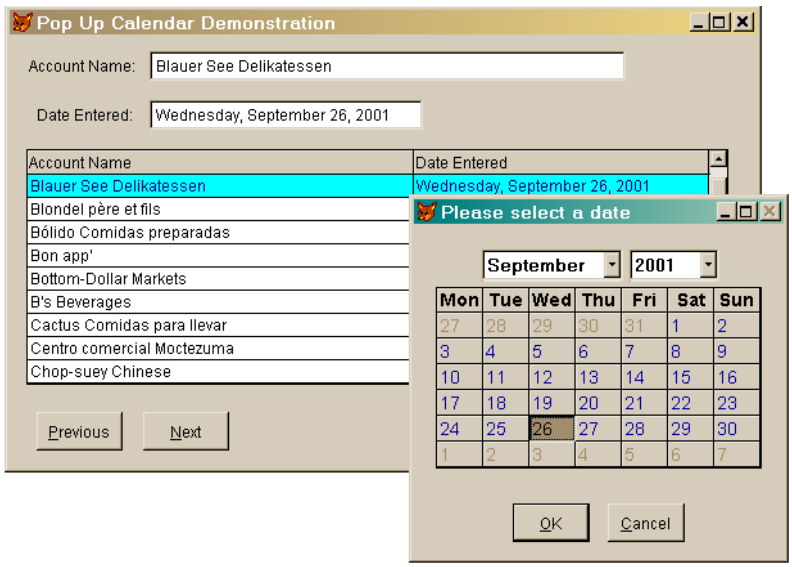


***Figure 2**. Pop-up calendar form in action.*

Notice, as shown in **Figure 2**, that the calendar form pops up immediately below the grid cell it is being called upon to update. This is no accident. The function that makes such intelligence possible is `OBJTOCLIENT()`, and it has been a part of the language since version 5.0. This function returns either a position or a dimension of the specified object relative to its form, depending on the parameter. We can obtain the coordinates at which to position our

pop-up calendar by determining the position of the textbox relative to the desktop. To do this, we first use **OBJTOCLIENT()** to obtain the form's position relative to the desktop. Then we use it again to determine the textbox's position relative to the form and add the two together.

To make it all work we added the custom *ShowCalendar()* method to the date textbox class and called it from the *DblClick()* method. The code in *ShowCalendar()*, listed next, instantiates the pop-up calendar form, passing it the coordinates at which the form should be positioned and the value with which the calendar should be initialized.

```
LOCAL luValue, lnTop, lnLeft

*** Calculate where the popup calendar should be instantiated
*** So it pops up directly below the date text box
*** SYSMETRIC( 9 ) is the height of the Form's title bar
lnTop = OBJTOCLIENT( Thisform, 1 ) + OBJTOCLIENT( This, 1 ) + ;
          This.Height + IIF( Thisform.TitleBar = 1, SYSMETRIC( 9 ) + 2, 2 )
lnLeft = OBJTOCLIENT( Thisform, 2 ) + OBJTOCLIENT( This, 2 )

DO FORM GetDate WITH lnTop, lnLeft, This.Value TO luValue
This.Value = luValue
```

The pop-up calendar, GETDATE.SCX, is a very simple modal form. Its custom *SetForm()* method, listed next, is called from the form's *Init()*. This method positions the form at the specified location and initializes the calendar with whatever date was passed to the form (the default behavior is to use today's date if nothing is specified).

```
LPARAMETERS tnTop, tnLeft, tdInitialDate
*** Initialize the combo with the passed date
*** Default to today if empty
WITH Thisform
  *** Position it correctly
  .Top = tnTop
  .Left = tnLeft
  *** Save the initial value so we can restore it
  *** if the user presses the cancel button
  .tInitialDate = tdInitialDate
  WITH .acxCalendar
    IF NOT EMPTY( tdInitialDate )
      .Object.Value = tdInitialDate
    ELSE
      .Object.Value = DATETIME()
    ENDIF
  ENDWITH
ENDWITH
```

When the user clicks the form's OK button, the following code populates the form's custom *tRetVal* property from the selected date in the calendar.

```
WITH Thisform
  .tRetVal = .acxCalendar.Object.Value
  .Release()
ENDWITH
```

Finally, this date is returned to the caller with this line of code in the pop-up form's *Unload()* method:

```
RETURN Thisform.tRetVal
```

How do I put a combo in a grid? (*Example: CH01.vcx:: cboGrdDropdown and ComboInGrid.scx*)

In Chapter 6 of KiloFox (page 190) we presented a combo class especially for use in grids. This class has a *style* of 0-Dropdown Combo, so users can type into the textbox portion of the control to add new entries to its *RowSource*. In order to prevent the user from doing this, the class has a custom *lAllowAddNew* property that may be set to false. Consequently, when *lAllowAddNew* is set to false, the user can still enter a new item in the combo only to be told "Please select an item from the list". This is user-surly behavior, to say the least.

There is also the issue of controlling the cursor keys. In a grid, we want the cursor keys to traverse the list when it is visible, but we want them to navigate the grid when the combo is closed. This is the default behavior of a drop-down combo and requires no additional code. It is only when we use a drop-down list in a grid that we need code to handle the cursor keys.

Because of these issues, we realized that it was a mistake to have a single "combo in grid" class to handle both types of combo boxes. What we really need is a drop-down combo class for use in a grid and a separate drop-down list class. The class presented in KiloFox works just fine as a drop-down combo and will continue to do so. Our task here is to create a special drop-down list class especially for use in a grid (see **Figure 3**).

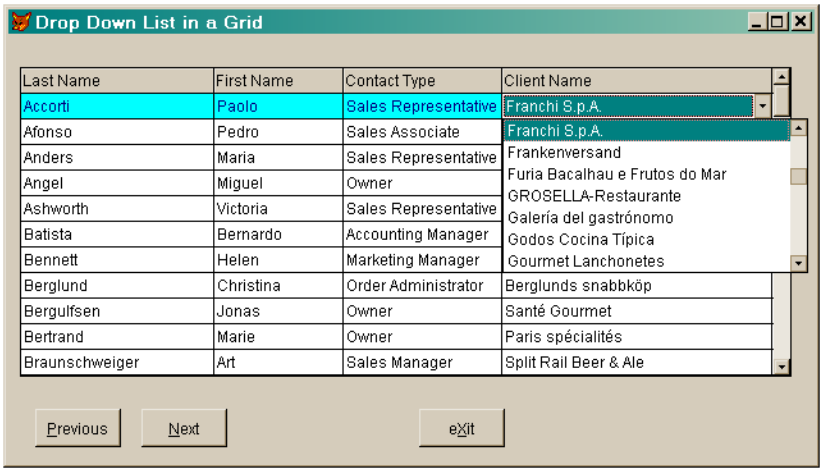


***Figure 3**. Drop-down list in a grid.*

Our drop-down list class has many of the same characteristics as the drop-down combo class presented in KiloFox. It has its visual characteristics customized for use in a grid: no border, plain style, and so on. It is this class that requires the custom *HandleKey()* method listed on page 194 of KiloFox. All references to this method can safely be removed from the

drop-down combo class and it will continue to function quite well as a drop-down combo in a grid.

To use the cboGrdDropdown class in a form, just drop it into the desired grid column and make it the column's *CurrentControl*. In our example, the combo is bound to the iClientFK field in the grid's *RecordSource.* Notice that we did not merely use a *RowSourceType* of 6-fields for our drop-down list and populate its *RowSource* as `Clients.cCompany, iClientPK`. There is a very good reason for this. If we used a *RowSourceType* of 2-alias or 6-fields for our combo, it would be blank when it got focus. This appears to be a bug that only occurs when you use a combo in a grid in this fashion, binding it to a foreign key value in the underlying data while displaying the descriptive text from a lookup table. Fortunately, with nine *RowSourceTypes* to choose from, we are able to work around this *gotcha!* easily enough.

The tricky part of using this class is getting the column's *ControlSource* set up correctly. Typically, the purpose of using a combo in a grid is to bind the column to some foreign key value in the grid's *RecordSource* while displaying the associated descriptive text from a lookup table. This means that when the column's *Sparse* property is left at its default value of false, all rows except the current one display the foreign key value instead of the descriptive text. In order to get around this problem in the sample form, we set the *Bound* property of the grid column named iColClientFK to false and its *ControlSource* to `( IIF( SEEK( Contacts.iClientFK, 'Clients', 'iClientPK' ), Clients.cCompany, '' ) )`. Now the client name is displayed in all rows of the grid.

At this point you are probably saying to yourself "Hang on, there! Setting up the column's *ControlSource* like that has the side effect of making the column *ReadOnly*!" and you would, in fact, be correct. However, when you run the example, you will see that you can still drop the list and make changes to the client for the current contact. The reason for this is that the *ReadOnly* attribute applies only to the portion of a control that accepts text. Since a drop-down list does not accept text, it cannot be made read only (any more than a command button could be made read only). Fortunately, this behavior is documented and is by design so it is unlikely to change in future releases of the product.

How do I run code when a projecthook is activated? (*Example: cPhkBase2.vcx::phkBase, phkDevelopment*)

In Chapter 15 of KiloFox (page 482), we noted a couple things missing from the first release of the projecthook class. The events missing from the projecthook were *Activate* and *Deactivate*. Microsoft rectified this with the release of Visual FoxPro 7.0.

We now have the ability to write code that fires each time the Project Manager gets focus. This allows us to perform a change directory to the project's home directory (KiloFox, page 492), set the path to the various directories used in the project (KiloFox, page 492), and change the IntelliDrop field mappings in the Registry to the base classes used for the project (KiloFox, page 494). Be careful to write optimized code in these methods; otherwise, you will see performance issues when activating a project. Also note that the project gets activated/deactivated quite frequently. Using the Project Manager to open any of the items that it contains deactivates the project. Closing the editor or designer reactivates the project if it is next in the window stack.

Another *gotcha!* to be aware of when writing code in the *Activate()* method is displaying a message via the `MESSAGEBOX()` function. It will cycle the deactivation/activation code because

the `MESSAGEBOX()` first deactivates the project and then reactivates it once the developer responds to the message. This interaction causes an infinite loop. If you have a messaging mechanism in the *Error()* method and an error in the *Activate()* method you can run into the same problem.

Things that we missed in KiloFox

Alas, no matter how hard we try we are always sure to miss something. People have generally been very nice about KiloFox, but there are a number of issues that have been mentioned over and over again as having been conspicuous by their absence from the first book. We would like to correct those oversights here and now.

How do I set focus to a control? (*Example: CH01.vcx::aFindObj, FindItDemo.scx*)

The trick to doing this is to remember that every object that can receive focus has both a *TabStop* and a *TabIndex* property. The first holds a logical value indicating whether the object should be included in the tab order for the container. When this property is set to false the user cannot access the control by using the Tab key. (Note, however, that it does not deactivate the control or prevent a user from giving the control focus by clicking on it.) The *TabIndex* property defines the sequence in which those controls whose *TabStop* property is set to true (the default value) are accessed when the Tab key is used.

How *TabIndex* works

When controls are added to a form, or any other container, Visual FoxPro automatically assigns the *TabIndex* to reflect the order in which the controls are added. Thus the first control has a *TabIndex* of 1, the second gets 2, and so on. Each container has its own internal tab order for the controls that it contains, and the container itself has an entry in its parent container's tab order. This means that at any level of containership there is a single tab order that applies to all objects that participate at that level. **Figure 4** shows a typical form, while **Table 2** shows how the different levels of containership affect the tab order.

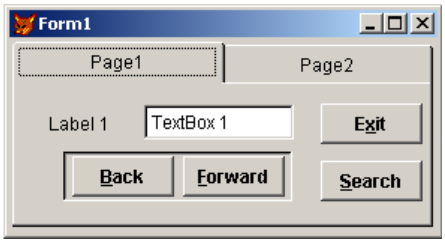


***Figure 4**. Form with nested objects.*

***Table 2**. Tab order in different levels of containership.*

Object	TabIndex	Applies to
Form		
PageFrame	1	Form
Label1	1	Page 1
TextBox1	2	Page 1
Container	3	Page 1
Back Button	1	Container
Fwd Button	2	Container
Search Button	4	Page 1
Exit Button	2	Form

Unless you are extremely well organized in constructing your forms and classes, you will need to amend these default values to ensure that the final form actually behaves in the way that your users expect. When in the Form (or class) Designer, the "View TabOrder" option from the Main Menu allows you to manually change the tab order and presents the current settings for the selected level of containership in either "Interactive" or "List" format (which you can specify on the Forms page of the Options dialog).

Finding the right control

To address the original question, to set focus to a specific control, all that is required is to loop through the container's collection of contained objects and find the right object. In all versions of Visual FoxPro prior to Version 7.0, each container has its own specific collection property—PageFrames have *Pages*, Pages have *Controls*, Grids have *Columns*, and so on. This means that, in earlier version, we always have to test the base class of a container and hard-code the correct collection name. Furthermore, each collection has its own, separate, counter property—for instance, *PageCount* or *ColumnCount*. In Version 7.0 all the container classes now have an *Objects* collection, which has an *Objects.Count* property, which makes writing this sort of code much simpler.

Notice that the Version 7.0 Help file appears to indicate that only certain classes have the Objects collection, but this is an error in this release of the documentation—all classes capable of containing other controls really do have one.

So, to set focus to a specific control, all we need to do is loop through the Objects collection until we find it. Then we can simply return the object reference to that control and set focus to it. The aFindObj class has three exposed methods (and four protected methods) that do exactly that, as shown in **Table 3**.

***Table 3**. Methods of the aFindObj class.*

Method	Parameters	Description
FindFirst()	Object	Uses the Protected *FindIndex()* method to return a reference to the first object in the passed container's tab order that can receive focus.
FindLast()	Object	Uses the Protected *FindIndex()* method to return a reference to the last object in the passed container's tab order.
FindObject()	Object Value to find Property to check	Searches the properties of all contained objects to find the first object that has the specified Property = Value pairing. Returns an object reference to that object if it can receive focus. Otherwise, returns the next object in the tab order that can receive focus.
FindIndex()	Object Index Number	Protected method that searches the tab order for a control with the specified index number in the passed container object and returns a reference to it if it can receive focus. Otherwise, returns a reference to the next object in the tab order that can receive focus.
PopulateArray()	Object Array	Protected method called from *FindIndex()* and *FindObject()* to populate a local array of contained objects that have a *TabIndex* property. The array is ordered by *TabIndex*.
SearchArray()	Array Index Number	Protected method called from *FindIndex()* and *FindObject()* to find the first object in the array (beginning at the current index into the array) that has a *SetFocus()* method. It also ensures that CommandGroups and OptionGroups are handled properly since one can only set focus to the contained buttons.
CanGetFocus()	Object	Protected method called from *FindIndex()* and *FindObject()*. Returns true if the passed control can receive focus.
ValidateObject()	Object	Protected method called from *FindIndex()* and *FindObject()* to ensure that an object reference to a container was passed to the class.

The first thing we need to be careful about is that, although objects that are based on the *Label* class have a *TabIndex* property, they have neither a *TabStop* property nor a *SetFocus()* method. This is because even though labels do not participate in the tab order and cannot actually receive focus, they can have "hot keys" assigned (by preceding the desired character with "\<" in the label's caption). When the hot key assigned to any label is pressed, the *next* control in the tab order is activated. (This behavior does assume that that labels always immediately precede, in the tab order, the control to which they relate.)

The second thing we have to consider is that, even though an object may have a *TabIndex* property as well as a *SetFocus()* method, it still may not be possible to set focus to it. If the object is invisible or disabled, we will need to find another object to which we can set focus.

Finally, there is the issue of CommandGroups and OptionGroups. These controls have a *TabIndex* property and, even though you cannot set focus to the button group, you *can* set focus to the contained buttons. Our class must handle all of these situations.

Using the aFindObj class (*Example: FindItDemo.scx*)

The *aFindObj* class is designed as a visual class and is intended to be dropped onto a form (although it will work with any container). It is based on the Custom Base Class because that class has no visual component and, more importantly in this case, has no *TabIndex* property and so will not interfere with the tab order. In order to use the class, drop it on a form. The

following code, from the *Activate()* method of the first page on the demonstrations form, sets focus to the first object in the tab order:

```
loObj = Thisform.oFind.FindFirst( This )
*** Be sure we got an object back !
IF VARTYPE( loObj ) = "O"
  loObj.SetFocus()
ENDIF
This.Refresh()
```

Similar code on the second page calls the *FindLast()* method to force focus to the last control. The two buttons call the *FindObject()* method in slightly different ways.

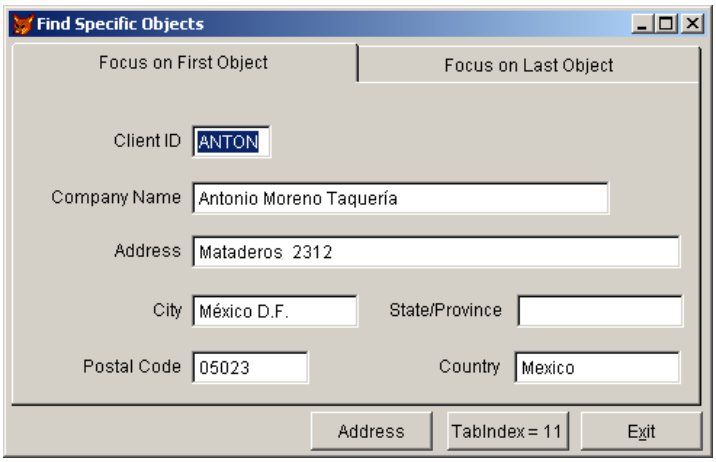


***Figure 5**. Form using the aFindObj class for setting focus to specific controls.*

The Address button in **Figure 5** requests a reference to the first object that has a caption property equal to "Address"—which just happens to be a label. The reference returned, as for a hot key, is therefore to the object that is next in the tab order. The other button simply requests a reference to whatever object happens to be eleventh in the tab order.

The two pages of this form actually display the same objects, but have a different tab order defined on each page. The result is that the Address button will select the field that is identified by the address label on both pages, but the other button selects a different object depending on which page is active.

How do I display the current record at the top of my grid? (*Example: TopGridRow.scx*)

In Chapter 6 of KiloFox, we showed you how to display the last full page of a grid for those special occasions where such specialized functionality may be required (page 173). However, we neglected to provide an example of how to make the current row the first visible row in the grid. This is an especially nice feature to have when you are searching for specific records in your grid. For example, it would have been very nice if the pop-up search form that we presented on page 370 of KiloFox had automatically positioned the located record at the top of

the grid. Somehow we overlooked that rather obvious refinement, so we are taking the opportunity to remedy it here.

The code that does the work is in the form's custom *Send2TopRow()* method, but it could just as easily be made a custom method of a grid class. The method makes the current row the first row by shrinking the grid until it displays only a single row. It then sets focus to the grid's *ActiveColumn* before restoring the grid to its original height, like this:

```
LOCAL lnHeight
*** Set LockScreen to avoid nasty visual side effects
Thisform.LockScreen = .T.
WITH Thisform.grdContacts
  lnHeight = .Height
  *** Set the grid's height
  *** so that we can "see" only the current record
  .Height = .HeaderHeight + .RowHeight
  Thisform.oActiveControl.SetFocus()
  *** When re reset the grid's height to what it was originally,
  *** the current record is automagically at the top of the grid
  .Height = lnHeight
ENDWITH
Thisform.LockScreen = .F.
```

How do I lock the leftmost column in my grid? (*Example: CH01.vcx:: GrdLockLeftColumn and LockGridColumn.scx*)

This is actually much easier to implement than you might think as long as you only want to keep the *first* grid column from scrolling out of sight when the grid is scrolled horizontally. Quite frankly, we can see a need for locking the first column when it contains information that identifies each row for the user as the grid scrolls horizontally. However, we feel that locking multiple grid columns defeats the purpose of using a grid. The primary reason for using a grid is to display a large amount of information to the user in as small a space as possible. By locking more than a single column you are severely limiting the viewable area as the grid is scrolled. You would be better served by displaying some "header" information for the current grid row on the form and synchronizing this information with the current grid row by refreshing the relevant controls in the grid's *AfterRowColChange* event.

The first thing we did to lock the leftmost grid column was to add a custom property to our grid class called *nFirstColumn*. Next, we had to determine which grid column would be the first column when the grid was instantiated. The following code, from the custom *SetGrid()* method, saved the index of this column to this custom *nFirstColumn* property. (Note: This index will usually be 1 *unless* the grid's columns have been rearranged in the form designer after they have been added to the grid—in which case it will be for whichever column now occupies the leftmost position.) What we are looking for here is the column whose *ColumnOrder* is 1, and this is not necessarily going to be the same as Grid.Columns[1].

```
LOCAL lnCol

DODEFAULT()
WITH This
  *** Find out which column is the leftmost column
```

```
  *** when the grid is instantiated and save it
  FOR lnCol = 1 TO .ColumnCount
    IF .Columns[ lncol ].ColumnOrder = 1
      .nFirstColumn = lnCol
      EXIT
    ENDIF
  ENDFOR
ENDWITH
```

The final piece is some code in our grid class's *Scrolled()* and *AfterRowColChange()* methods. The code must be called from both places because even though tabbing horizontally through the grid may indeed scroll it horizontally, this action does not cause its *Scrolled* event to fire. The only difference between these two methods is the **IF** condition used to determine whether we need to reset the *ColumnOrder* of the column that we want to lock.

This code in the grid's *AfterRowColChange()* method uses the grid's *LeftColumn* property to keep the locked column from scrolling out of sight when the grid is scrolled horizontally. *LeftColumn* contains the *ColumnOrder* of the leftmost column in the visible portion of the grid.

```
WITH This
  *** if we scrolled horizontally, adjust the column order
  *** so the leftmost column stays the leftmost column
  IF .RowColChange > 1
    .Columns( .nFirstColumn ).ColumnOrder = .LeftColumn
  ENDIF
ENDWITH
```

Did you notice the reference to the grid's *RowColChange* property? This is a new feature in Visual FoxPro 7 that lets us determine whether the column, the row, neither, or both has changed.

How do I create truly generic command buttons? (*Example: CH01.vcx::cmdAction and CH01.vcx::frmSample and Navigate.scx*)

Command buttons are "action" controls. When the user clicks on one, the expected behavior is an action of some sort, whether it is closing a form, launching a form, or printing a report. In KiloFox, we created a generic command button class that had a custom *onClick()* method into which all custom code was written (page 120). Here we have taken the idea one step further.

The premise is that the function of a command button is to notify its container that it has been clicked. It follows, therefore, that the action method code should reside in that container. However, in order to keep the code in our command buttons generic, we make the button class look for, and call, a standard method in its immediate container, named *DoAction()*. The command button passes to that method the name of the method that should be executed (this is held in a custom *cAction* property of the button). If the parent container does not have a *DoAction()* method, the button will call upon the form's *DoAction()* method.

Since we are talking about command buttons here, it is very safe to assume that they are ultimately resident on a form. This code, in the custom *onClick()* method of our command button class, executes the method call:

```
*** Tell the parent container that I have been clicked
*** If the parent container has a DoAction method call it
*** Otherwise, tell the form
IF PEMSTATUS( This.Parent, 'DoAction', 5 )
  This.Parent.DoAction( This.cAction )
ELSE
  Thisform.DoAction( This.cAction )
ENDIF
```

This design results in "no muss, no fuss" command buttons. In order to use them, all you need to do is drop them in a container and set their *cAction* property to the name of a method to execute. The only other required code is that the custom method specified by the button's *cAction* property should exist somewhere in the containership hierarchy. Of course, the assumption in all of this is that all root classes that can contain other objects adhere to the public interface that we have defined; that is, they all have a custom *DoAction()* method.

The essence of the *DoAction()* method, at any level of containership below that of the Form, is that it checks to see whether it has a method whose name matches that which is being requested and, if so, executes it. If it does not have such a method, it tries to pass the call on to its own parent, if that object has a *DoAction()* method, and if not, it passes the call directly to the form.

```
LPARAMETERS tcMethod
LOCAL lcMethod, luRetVal
*** Do we have that method available
IF PEMSTATUS( This, tcMethod, 5 )
  lcMethod = 'This.' + tcMethod + '()'
  luRetVal = &lcMethod
ELSE
  *** We don't have one of those here, try immediate parent
  IF PEMSTATUS( This.Parent, 'DoAction', 5 )
    luRetVal = This.Parent.DoAction( tcMethod )
  ELSE
    luRetVal = Thisform.DoAction( tcMethod )
  ENDIF
ENDIF

RETURN luRetVal
```

At the form level, there is no further possible "parent" so the Form's *DoAction()* method merely displays a standard "under construction" message when a method is not available so that work in progress does not blow up during testing.

```
LPARAMETERS tcMethod
LOCAL lcMethod, luRetVal

IF PEMSTATUS( This, tcMethod, 5 )
  lcMethod = 'This.' + tcMethod + '()'
  luRetVal = &lcMethod
ELSE
  MESSAGEBOX('Coming soon to a computer near you...',64,'Under Construction')
ENDIF
RETURN luRetVal
```

If you think that all this looks rather familiar, you are right—it is actually one form of a "*Chain of Responsibility*" Pattern.

How do I set up a hot key to declare local variables? (*Example: DeclareLocals.prg*)

One of the strengths of Visual FoxPro is that it does not enforce strong typing of variable declarations. However, this is a double-edged sword. If we don't explicitly declare a variable, or misspell one that has already been declared, VFP automatically creates a new one for us that is, by default, private in scope. This behavior can actually introduce bugs that are really difficult to track down. Wouldn't it be nice if we could set up a hot key that would automatically search for a **LOCAL** declaration, create one if it does not already exist, and add the word under the cursor if it is not yet declared?

In the past, Cobb Editor Extensions, a public domain add-on for VFP, provided us with this functionality. In VFP 7, IntelliSense provides us with most of the functionality that we used to get from CEE (see Chapter 3 for more detailed information on configuring IntelliSense), but not this specific feature. Unfortunately, Cobb interferes with IntelliSense, so the choice is either use IntelliSense and not have automated **LOCAL** variable declaration, or stay with Cobb and not be able to use IntelliSense. Neither of these is satisfactory, so what can we do about it?

FoxTools is a Visual FoxPro API library that has been around for many years and exposes Windows DLLs for use in Visual FoxPro. Functions in the FoxTools library allow us to set and retrieve file information, manipulate paths and file names, use system alerts, and perform many other functions. Many functions that were, originally, only available through FoxTools (for example, **JUSTSTEM()**, **JUSTEXT()**, and **ADDBS()**) were also added to the native language in Version 6, and Version 7 has added more. **GETWORDCOUNT()** does the same thing as the **WORDS()** function in the FoxTools library: It returns the number of words in the passed string. The new **GETWORDNUM()** function has the same functionality as the FoxTools **WORDNUM()** function: It returns the specified word from a string given the string and the index of the word to retrieve. Because so much of the functionality from this API library is now a part of the language natively, we may be tempted to forget all about FoxTools. This would be a mistake. For example, there are 33 FoxPro editor API functions that are exposed by FoxTools, and we can use them to implement the functionality that used to be provided by CEE.

The first thing that is required is to define the set of characters that can signify the beginning or end of the word under the cursor. Some of these, like a space or tab, are fairly obvious. Some, like the arithmetic operators and parentheses, are not. The set of delimiters that we came up with is:

```
#DEFINE WORDDELIMITERS "!@#$%&*()-+=\[]:;<>?/,. "+CHR(9)+CHR(13)+CHR(10)
```

All of the FoxPro editor API functions require the wHandle of the window that is being manipulated. This line of code retrieves the wHandle of the foremost window:

```
lnHandle = _WonTop()
```

However, just because a window is foremost does not mean that it is a code editing window. In order to make that determination, we make use of the *_EdGetEnv()* function, like this:

```
lnResult = _EdGetEnv( lnHandle, @laEnv )
```

This populates the *laEnv* array with 25 individual items of information about the current editor. If successful, the function returns a value of 1 and if it fails, it returns 0. We are only interested in the items in the array shown in **Table 4**.

***Table 4**. Key items returned by _EdGetEnv().*

Index	Item description	Defined values
1	File Name	
2	File Size	
12	Read Only?	0 – No 1 – Yes 2 – File is read-write but was opened read-only 3 – File is read-only and was opened read-only
17	Selection Start	
18	Selection End	
25	Editor Session	0 – Command window 1 – Program file opened with MODIFY COMMAND 2 – Text file opened with MODIFY FILE 8 – Menu code edit window 10 – Method code edit window 12 – Stored procedure in a DBC

We use this information to determine whether or not there is anything for our *DeclareLocals* program to do. If the editor window is not of the correct type (that is, a code editing window), or if the file is empty or read-only, there is no reason to do any more processing.

```
IF ( lnResult = 0 ) OR;
   ( laEnv[ 2 ] = 0 ) OR ;
   ( laEnv[ 12 ] > 0 ) OR ;
   ( laEnv[ 17 ] = 0 ) OR ;
   ( NOT INLIST( laEnv[ 25 ], 1, 8, 10, 12 ) )
  RETURN
ENDIF
```

Having determined that we are in a code editing window, we have a number of tasks to tackle:

- Isolate the word that is under the cursor or that is selected. This is handled by the *GetVariable()* function.
- Retrieve the entire text into an array (Warning: This is, therefore, limited to a maximum of 65,000 lines) and save the current line number so that we can get back to the correct place later.

- Search back through the code, handled by the *GetLocalInfo()* function, looking for (in the following order of precedence):
    - A **LOCAL** declaration but ignoring **LOCAL ARRAY** declarations
    - A Parameters statement (either **LPAR** or **PARA**)
    - A Function or Procedure statement (either **FUNC** or **PROC**)
    - The beginning of the current file
- If an existing **LOCAL** declaration is found, check *all* declarations for the current variable (to avoid double declarations). This is handled by the *IsDeclared()* function, which, if the variable already exists, displays a message to that effect and returns True otherwise.
- Insert a new line, with a new **LOCAL** declaration, at the line immediately after the position at which the search through the code stopped. This is handled by the *InsertLocalDeclaration()* function, if, and only if:
    - No **LOCAL** declaration has been found
    - An existing declaration is longer than 100 characters (this is a purely arbitrary limit in the interest of readability only)
- Add the variable name under the cursor to the **LOCAL** declaration statement, preceding it with a comma if applicable, and display an appropriate message. This is handled by the *UpdateLocalDeclaration()* function.
- Re-position the cursor at the starting location and exit.

These steps are managed by the main body of the program like this:

```
*** Get the current cursor position
lnSelStart = laEnv[ 17 ]

*** Get the variable (if there is one) at the insertion point
lcVarName = GetVariable( lnHandle, lnSelStart, @laEnv )
IF EMPTY( lcVarName )
  *** Nothing to do
  RETURN
ENDIF

*** Get the contents of the editing window into an array
*** beware! if you have more than 65,000 lines of code,
*** this will crash
lcProgText = _EdGetStr( lnHandle, 0, laEnv[ 2 ] - 1 )
lnLines = ALINES( laLines, lcProgText )

*** Get the line number ( the number returned is 0-based )
*** in which the cursor is currently positioned
lnCurLine = _EdGetLNum( lnHandle, lnSelStart )

*** Now see if we have a "local" declaration already
*** And if we don't get the line number at which to insert one
loLocalInfo = GetLocalInfo( @laLines, lnCurLine )
```

```
IF VARTYPE( loLocalInfo ) # 'O'
  *** Mayday! MayDay! We are Fubar!
  RETURN
ENDIF

lnLine = loLocalInfo.nLineNo

*** See if we already have a local declaration
*** if we do, we are going to have to check for the
*** variable already declared
IF lolocalInfo.lLocal
  IF NOT( IsDeclared( lcVarName, @laLines, lnLine ) )
    *** If the current declaration is longer than 100 characters
    *** Start a new LOCAL declaration line
    IF LEN( ALLTRIM( laLines[ lnLine ] ) ) > 100
      InsertLocalDeclaration( lnHandle, lnSelStart, lnLine, lcVarName )
    ELSE
      UpdateLocalDeclaration(lnHandle, lnSelStart, @laLines, lnLine, lcVarName)
    ENDIF
  ENDIF
ELSE
  *** This is the first local declaration
  InsertLocalDeclaration( lnHandle, lnSelStart, lnLine, lcVarName )
ENDIF
```

To use DeclareLocals.prg in your development environment, copy it to your VFP root directory and add this line to the program that configures your development environment (using your personal favorite hot key):

```
ON KEY LABEL ALT+6 DO DeclareLocals.prg
```

If you aren't using a program to configure your development environment (and why on earth not?), just type the same line directly in the command window.

This program makes the assumption that you declare your variables as comma-separated strings. Furthermore, each line of the declaration must start with its own LOCAL key word (that is, no continuation characters may be used). Failure to adhere to this convention will result in anomalous behavior.

If you have highlighted an entire word (for example, m.lcVariable), pressing Alt-6 will insert the entire highlighted text (m.lcVariable) into your LOCAL declarations. If you prefix your local variables like this, do ***not*** *highlight the entire word. Just press Alt-6 when the cursor is at the end of the variable and it will be declared without the "m." prefix since the "." is defined as a word delimiter. This also has the advantage of being declared something like laArray[10, 2] as LOCAL laArray[10, 2] if you highlight the array* ***and*** *the dimensions before pressing Alt-6. However, in most cases, you do* ***not*** *want to highlight the entire word want to be declared.*

Chapter 2
Data Driving with VFP

This chapter provides examples of various ways you can utilize the power and flexibility of Visual FoxPro to data drive your applications. There are many benefits to data driving, but perhaps the single most important one is the ability to change functionality without actually having to re-compile the application code. In these days of Internet applications, this is probably one of the most compelling reasons to use Visual FoxPro as your primary middle-tier development tool.

What exactly is "data driving"?

Data driving is the name given to the design technique in which the actual code written in an application is generic and relies on information stored outside of the code in order to deliver the required behavior at run time. Visual FoxPro is particularly good for working with this sort of code because it has the ability to evaluate name expressions and perform macro substitution at run time and therefore makes it possible to write code like this:

```
LPARAMETERS tcAlias
IF USED( 'sourcefile' )
  *** We have this alias open, so close it
  USE IN sourcefile
ENDIF
*** Now open the one we want
USE (tcAlias) IN 0 AGAIN ALIAS sourcefile
*** All code from here on refers to the table as 'sourcefile'
```

This is a very simple example of one form of data driving with which you are probably perfectly familiar, although you may not think of it as "data driving". What this code allows us to do is to refer only to the alias name "*sourcefile*" in the remainder of the code. We no longer need to know the actual name of the table that is being used as "*sourcefile*" and we can be sure that the code will work correctly with any table whose structure does not conflict with explicit references to fields or their data types. While code of this sort is most often used to handle data from tables that share a common structure, it can still work with tables whose structures are not identical, either by ensuring that only common fields are referenced or by including appropriate conditional tests in the code.

"Hang on!" you are probably thinking, "This isn't really data driving, this is merely parameter driven code." and you would be right. A parameter is, after all, merely an item of data. So the question is, where does the parameter come from—and to understand that, we need to understand a bit more about the types of data that exist.

The three different types of data

As Visual FoxPro developers we are accustomed to dealing with data without thinking too much about where it comes from or what it actually represents. In fact, there are three distinct types of data with which we have to deal: core, process, and metadata.

Core data

Core data is the fundamental information that a business needs in order to carry out its functions. Such data cannot be derived from existing data and must be directly entered into the system, although the mechanism by which the data is entered is irrelevant. It could equally well be from a user typing at the keyboard as through some form of automated data download or import.

Examples of core data include such things as the Look-Up tables used by an application (anything from Type Codes to the "Chart of Accounts" table) or the name, address, and similar personal information held about Customers and Suppliers.

Process data

Process data is information that is used by a business as part of its function, and so must be captured and stored, but that can be derived from other, pre-existing, data. It does not have to be entered directly into the system and is usually the result of processing other items of information—hence the name.

Examples of process data include such things as the Line and Invoice Totals in an order processing system. Clearly the first is derived from the multiplication of Quantity Ordered by Item Price (both of which are *core data* items), while the second is calculated from the sum of all line totals (which are themselves *process data*) modified by the application of Tax, Shipping, and Discount values (a mixture of both core and process data). All of these elements must be stored because without them, recalling a past invoice would have to apply "current" values for taxes, shipping costs, and so on. It is likely that these values have changed over time and are different from those that applied when the invoice was first raised. Clearly, it would be very inconvenient to have our invoice totals changing after they have been closed.

Metadata

Metadata has often been defined as "*data about data*". However, perhaps a more usable definition is that metadata is data that does not contain information that is required by the business in order to fulfill its functions. Its role is to manage and control the behavior of a data driven system and, generally, the end users never see, or even know about, the metadata.

As a Visual FoxPro developer you are already familiar with several types of metadata because Visual FoxPro itself uses it extensively. For example, the database container, which contains information required to manage tables, is truly "data about data". Another good example is the FRX file used by the Visual FoxPro report writer. This file is actually a standard FoxPro table that is used by the report writer to determine *how* it should produce output. As a matter of fact, you can `USE` an FRX and `BROWSE` it just as you can any other table

What goes into the metadata?

The answer to that question depends upon the nature of the functionality that you wish to data drive. In the example quoted at the start of this section, the parameter we need to pass to the code is the name of a table. The metadata to drive that code would, therefore, consist of some form of lookup to relate a key value, which is meaningful to the user, to the name of a specific table that is probably a level of detail that the user does not need, or even want, to know about.

The examples that comprise this chapter show how different applications of the data driving technique use metadata differently but, in the end, they all come down to some form of lookup.

Where should metadata be stored?

The most obvious place for us, as Visual FoxPro developers, is in a Visual FoxPro table. Tables (along with cursors and views) are what Visual FoxPro is built to handle best, and it has a superb range of tools for dealing with any aspect of metadata management.

Most "back-end" databases store data in "*pages*" and employ "*set-based*" Structured Query Language (SQL) to interrogate the data. Visual FoxPro, on the other hand, is record-based. It is designed for processing a sequential set of records and has commands and functions that are optimized for that purpose. This makes it ideally suited for handling the rapid, very specific, look-up and retrieval functions that are required when data driving an application.

However, Visual FoxPro tables are by no means the only place to store metadata. Other possible mechanisms that may be appropriate under certain circumstances include Cascading Style Sheets and XSL or INI files or the Registry.

Style Sheets and XSL

Essentially these provide data-independent methods for driving the display of HTML (Style Sheets) or XML (XSL) in Web pages. By referencing the appropriate mechanism when defining pages that require data, the functionality for managing the display can be separated from the data itself. In this respect they both meet the definition of metadata in the sense that they contain information about data. We shall have more to say about their use in Chapter 16, "Using Visual FoxPro on the World Wide Web".

INI files and the Windows Registry

At first sight these may seem like very different animals, but in fact they suffer from the same basic limitation when used to store metadata because they are both designed for storing and retrieving information in the form *attribute* = *value*. This is entirely appropriate given that their usual function is to handle configuration and setup information. While they are not good vehicles for storing processing information or actual code, each has specific advantages and disadvantages.

INI files are simply formatted text files and so can be edited using any text editor. Even relatively inexperienced end users easily comprehend their structure and function, and so they are most appropriate when application settings need to be maintained by end users themselves. We included, in KiloFox, a class for working with INI files in an application (see Chapter 10, page 313).

Although the Registry is more difficult to work with, it provides significant advantages over INI files when dealing with setup information that needs to be available system-wide or when access to Windows-specific functionality (for example, User Profiles) is required. A set of classes for working with the Registry is included with the Visual FoxPro Foundation classes (see REGISTRY.VCX).

Why bother with data driving?

Probably the single biggest benefit of using data driven components in your applications is that it minimizes the need for code changes. When circumstances change, as they invariably will, all that is needed is to change the affected entries in your metadata tables. There is no need to re-compile code, or take an application off-line so that updates can be made. This is vital when you are dealing with applications that run 24 hours a day, seven days a week or if you are developing applications for the World Wide Web. (You do not want to be taking down your Web server each time the client requests a minor modification to the application.)

A secondary benefit is that data driving requires that you keep the code in your application components generic. This means that they are much more likely to be truly reusable than if the specific functionality is hard-coded directly into procedures or method code. One of the biggest, and most common problems that we encounter when designing classes is that specific functionality gets included too high in the class hierarchy. By keeping code generic, and data driving the specific, you are much less likely to fall into this trap.

However, as we all know, there is no such thing as a free lunch; there must be some disadvantages of data driving an application. We can think of three. First, there is the performance overhead involved; second, the problems associated with designing a data driven application; and third, the issues associated with maintaining such an application.

Performance overhead

As we have already stated, the basic concept that underpins data driving is reliance upon some form of look-up. It doesn't really matter whether that look-up is into a local table, an INI file, or an external source like a Style Sheet, there are two sources of delay involved. First, it will take a finite (albeit very small) amount of time to execute the look-up and determine the result. Second, because the code then has to interpret the results of the look-up, there is an additional, also small, delay when compared to the situation where the code is explicitly written into the procedure or method.

However, these delays are generally very small and, especially when dealing with local Visual FoxPro tables, will generally be undetectable in the context of a working application. But they are real, and whenever you are considering using a data driven methodology you must take account of, and test for, these inherent delays.

Design considerations

Designing a data driven application, or application component, is considerably more difficult than simply writing explicit code to deliver the required functionality. First there is the issue of determining just how the data driven components are going to be integrated with the application as a whole. Second, and much more difficult, is the issue of designing and future-proofing the code.

The code written in a conventional application represents a snapshot view of the functionality that was deemed required when the code was written, which is why changes in requirements inevitably require changes in code. However, when writing the code for a data driven component, the objective is to move the specific functionality out of the code itself. Writing generic, reusable code is always harder than writing explicit, one-off code.

The design of the supporting data presents another set of issues. Having to make changes to the structure of the data source used to drive the code is something that has to be avoided at all costs since it would certainly mean changing the code as well—which defeats the whole purpose of data driving in the first place. On the other hand, who can predict how an application may evolve over time? The solution you will see being used in the examples in this chapter is to ensure that when we are designing the supporting tables we include a general-purpose memo field (usually named "*mproperties*") from the very beginning.

Maintenance issues

This one may be a little surprising but it can be a very real problem. Interpreting what a block of code is doing is something that we are all familiar with, and like all good developers, we ensure that we write comments when needed to explain what code is doing, and why a particular piece of code has been built in a particular way.

However, since the code is using metadata, what is happening may not be readily apparent to other developers working with it. In fact, it may not even be readily apparent to us when we revisit this code in six months! Copious, clear, and very descriptive comments when we create the code are the only solution that we know of to this problem. Documentation is important, but comments are critical!

So is data driving worth it?

The answer, in our opinion, is an unequivocal yes. The rest of this chapter is dedicated to providing working examples of how data driving various applications, or application components, can greatly simplify the task of maintaining your code.

How do I data drive my menus?

Before we attempt to answer that question, let us just review the capabilities of Visual FoxPro where menus are concerned. In fact we already have, and always have had, a fully data driven menu generator in Visual FoxPro. However, apart from some minor enhancements to give menus the same look and feel as standard Windows menus, it has not changed much since FoxPro Version 2.6. The data produced, and used, by the Menu Designer is stored in a table with an associated memo file (the MNX/MNT files). This table is used by the *GenMenu* program to create an MPR file, which in turn has to be compiled as an MPX file before the menu can be run. In short, although it is undoubtedly data driven, the implementation is certainly not very responsive to change and feels rather cumbersome.

This is odd because one of the most noticeable changes in the Windows environment over recent years has been the increasing use of context-sensitive pop-up menus, typically initiated by the user right-clicking their mouse over a particular control. In Visual FoxPro we can create pop-up menus in the Menu Designer by choosing "Shortcut" from the option dialog that appears when a `CREATE MENU` command is issued (see **Figure 1**). However, from that point on the process is identical to creating any other menu and is just as tiresome.

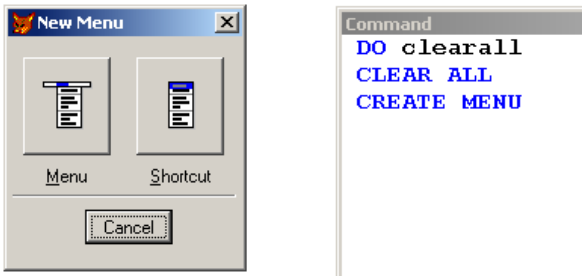


***Figure 1**. Menu creation options.*

What type of menus do we want to data drive?

In reality, there are fundamental differences between the way a shortcut menu and a menu bar are used, which are not reflected in the way in which they are created and maintained. In general, the system-level bar is normally used to control the application as a whole. It is, therefore, not unreasonable to assume that changes to the menu will only be needed when there are changes to the basic application code (that is, new or changed functionality). Such changes will, of course, necessitate re-compiling the application anyway. Under those circumstances, the rather cumbersome process needed to make the changes to the menu is not particularly onerous, as it is only one small part in the process of updating and regenerating the application.

On the other hand, a pop-up, or shortcut, menu is generally used to give access to some existing piece of functionality from different places in an application or to allow a user to choose from a list of available options. Changes to these types of menus are not likely to occur as a consequence of major changes to the application. More often they are required because users ask for an existing function to be made available at a new location, or the accessibility to a particular function has changed (either broadened, or restricted). Having to re-compile the entire application merely to add an option to a single pop-up menu, or to change a **`SKIP FOR`** clause, does not strike us as very efficient.

The conclusion we reached is that there was little point in our trying to re-write the entire menu generation process, but that we could do something to simplify the maintenance of shortcut menus. All that we have to do is to generate and execute an MPR file for our shortcut on the fly. One of the new functions in Version 7.0 is directly relevant here. The **`EXECSCRIPT()`** function allows us to run a block of code that is held in a memo field, or a memory variable, without the need to create and compile a temporary program file. So all that is needed now is somewhere to store the metadata for our shortcut menus and a class to generate and run the menu.

MPR file structure for a shortcut menu

If we examine the MPR file that is generated for a shortcut menu we can see that it is very simple indeed. It consists of only four components and, of these, the first and last are identical in all shortcuts except for the insertion of the appropriate name.

- A pop-up definition statement in the form:
  `DEFINE POPUP <name> SHORTCUT RELATIVE FROM MROW(),MCOL()`

- A series of definitions using the **DEFINE BAR** command (which includes the optional **SKIP FOR** clause)
- A series of Action definitions using **ON SELECTION BAR** command
- An activation statement in the form:
  ACTIVATE POPUP <name>

To create our own data driven shortcut menu handler, we need to define a data structure to store the information required for generating the menu bars and their associated action and skip for conditions.

The shortcut menu metadata

In order to maximize the reusability of menu bar definitions, we use a simple relational structure to create a many-to-many relationship between the menu name and the bars to be associated with that name (see **Figure 2**).

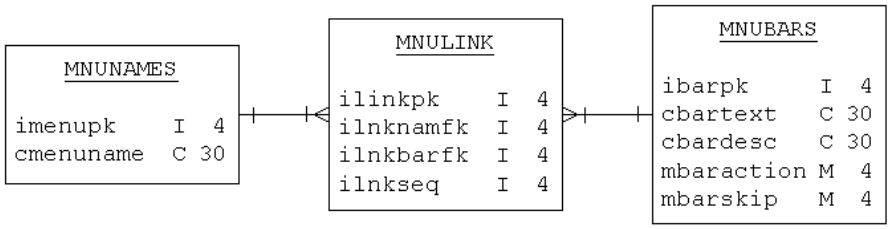


***Figure 2**. Shortcut menu metadata tables.*

The first table (MNUNAMES.DBF) is simply the header table that will be used to identify individual shortcut menus and that defines the key names that will be used to look up the bars for each menu. Note that these names cannot contain any spaces.

The link table (MNULINK.DBF) is the allocation table that allows us to implement a many-to-many relationship between the menu bar definitions and the menus that use them. This minimizes the number of bars that we need to define and allows us to reuse bar definitions in multiple menus. By keeping the sequence number in the link table, we ensure that the same bar can be used in different positions in several menus.

The final table in the triad (MNUBARS.DBF) is used to store the bar definitions. Each definition is comprised of a primary key and four fields: the actual text to be displayed in the shortcut menu (*cBarText*), a description for that text (*cBarDesc*), the code that is to be executed when the bar is selected (*mBarAction*), and, optionally, the code for the **SKIP FOR** clause (*mBarSkip*). The code that is entered into these last two memo fields will be parsed out line by line and passed as a "[]" delimited string to the **EXECSCRIPT()** function by the shortcut menu generator class.

NOTE: To avoid errors in compiling the scripts, it is imperative that you do not use the "[" and "]" characters in any code that you enter into these fields.

Although this class and its attendant metadata is intended for developer use only (and therefore we could expect people just to use a simple browse), we have included three forms that can be used to maintain these tables in the sample code for this chapter. The first, POPMENU.SCX (see **Figure 3**), is the control form that allows shortcut menus to be created or edited. POPNAMES.SCX (see **Figure 4**) is the pop-up form that updates MNUNAMES.DBF, the table that holds the menu names. POPBARS.SCX (see **Figure 5**) is called upon to manage the MNULINK.DBF and MNUBARS.DBF tables. These two tables contain the details of the individual menu bars. MNUBARS.DBF holds the details for the individual menu bars (command, skip for condition, and so forth). MNULINK.DBF links individual menu bars to specific menus so that a single entry in MNUBARS.DBF can be used in multiple menus.

This form includes a Run button to allow you to check the appearance of menus as they are defined. Notice also that the Delete button refers only to the entry in the link table—it does not delete the bar definition.

Sequencing the bars in a menu is handled by specifying either the preceding or succeeding bar by selecting from the lower drop-down list.

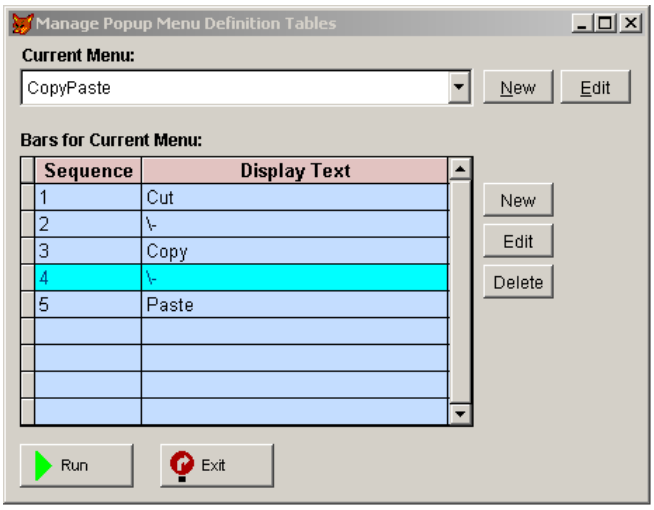


***Figure 3**. Shortcut menu manager.*

***Figure 4**. Shortcut menu name management form.*

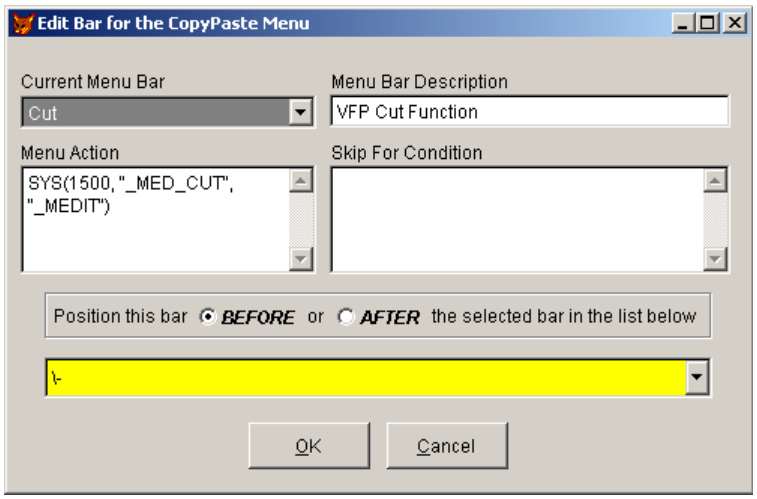


***Figure 5**. Shortcut menu bar management form.*

The shortcut menu generator class (*Example: PopMenu.prg::xPopMenu*)

The class that handles the generation of the shortcut menus on the fly is based on the native "*Session*" base class so that the metadata tables can be given their own private datasession and so will not interfere with anything that may already be running. All of the code is run directly from the *Init()* method which expects to receive, as a parameter, the name of the menu to be generated.

Running the code from the *Init()* method means that we do not actually need to create a reference to the object—once the menu is finished, we can just release it by returning False. The *Init()* calls the custom *GetMenuDef()* method, which determines whether a menu has been defined in the metadata tables for the passed-in name using SQL to populate a local cursor:

```
PROTECTED FUNCTION GetMenuDef(tcMenuName)
  LOCAL lcMenuName
  lcMenuName = UPPER( ALLTRIM( tcMenuName ))
  *** Populate the cursor
  SELECT MB.cbartext, MB.mbaraction, MB.mbarskip, ML.ilnkseq ;
    FROM mnunames MN, mnubars MB, mnulink ML ;
   WHERE MB.ibarpk = ML.ilnkbarfk ;
     AND ML.ilnknamfk = MN.imenupk ;
     AND UPPER( ALLTRIM( MN.cmenuname ) ) == lcMenuName ;
     AND NOT DELETED( 'mnulink' ) ;
    INTO CURSOR curMenu ;
    ORDER BY ML.ilnkseq
    *** Did we get anything?
  RETURN (_TALLY > 0)
  ENDFUNC
```

If the query returns a definition, the custom *BuildMenu()* method is then called to populate a local *lcScript* variable in the *Init()* method. The *BuildMenu()* method simply calls on two other methods, *GetBars()* to generate the `DEFINE BAR` statements:

```
PROTECTED FUNCTION GetBars( tcMenuName )
LOCAL lcBardef, lcTxt, lcBarNum, lcSkip
  *** Preamble here
  lcBarDef = "DEFINE POPUP " + tcMenuName ;
           + " SHORTCUT RELATIVE FROM MROW(),MCOL()" + CRLF
  SELECT curMenu
  GO TOP
  SCAN
    *** Prompt here
    lcTxt = "[" + ALLTRIM( curmenu.cbartext ) + "]"
    *** Sequence Number
    lcBarNum = TRANSFORM( curmenu.iLnkSeq )
    *** Skip For
    IF NOT EMPTY( curmenu.mbarskip )
        lcSkip = "SKIP FOR " + ALLTRIM( curmenu.mbarskip )
    ELSE
        lcSkip = ""
    ENDIF
    *** Definition
    lcBarDef = lcBarDef + "DEFINE BAR " + lcBarNum ;
             + " OF " + tcMenuName ;
             + " PROMPT " + lcTxt ;
             + lcSkip + CRLF
  ENDSCAN
  RETURN lcBarDef
ENDFUNC
```

and *GetActions()* to generate the **ON SELECTION BAR** commands:

```
PROTECTED FUNCTION GetActions( tcMenuName )
LOCAL lcScript, lcBarNum, lcAction
  lcScript = ""
  SELECT curMenu
  GO TOP
  SCAN
    *** Do we have an action defined
    IF EMPTY( curmenu.mbaraction )
      LOOP
    ENDIF
    *** Sequence Number
    lcBarNum = TRANSFORM( curmenu.iLnkSeq )
    *** Action
    lcAction = "[" + ALLTRIM( curmenu.mbaraction ) + "]"
    *** Need to embed all variants for CRLF chars
    lcAction = STRTRAN( lcAction, CHR(13)+CHR(10), "] + CHR(13)+CHR(10) + [" )
    lcAction = STRTRAN( lcAction, CHR(10)+CHR(13), "] + CHR(13)+CHR(10) + [" )
    lcAction = STRTRAN( lcAction, CHR(13), "] + CHR(13) + [" )
    lcAction = STRTRAN( lcAction, CHR(10), "] + CHR(10) + [" )
    *** Build the statement
    lcScript = lcScript + "ON SELECTION BAR " + lcBarnum ;
             + " OF " + tcMenuName ;
             + " EXECSCRIPT( " + lcAction + ")" + CRLF
  ENDSCAN
  RETURN lcScript
ENDFUNC
```

The result is that a script, containing all the code that would normally be stored in the MPR file, is returned to the *Init()* method as a string. The native `EXECSCRIPT()` function is then called to run the script, on completion of which the object releases itself.

Using the shortcut menu class (*Example: frmScut.scx*)

To invoke a menu, all that is needed is to instantiate an object based on this class and pass it the name of the shortcut menu required as follows:

```
NEWOBJECT( 'xPopMenu', 'popmenu.prg', NULL, 'copypaste' )
```

There are several ways of utilizing this functionality, but the one that we particularly like uses a textbox class with a custom *cMenu* property. This is used to hold the name of the menu to be invoked. Code in the *RightClick()* method of the class checks this property and, if it is not empty, invokes the specified menu. The first textbox on the sample form (see **Figure 6**) is an instance of this class that calls a simple Cut/Copy/Paste shortcut menu. (Notice that the Paste option has been set up so that it is disabled when the clipboard is empty.)

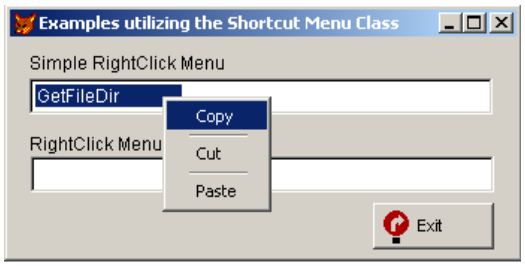


***Figure 6**. Shortcut menu bar example (frmscut.scx).*

One of the drawbacks of using shortcut menus is that they cannot directly return a value to the calling code. The solution is to use a variable that is scoped as Private in the calling method and have the code called by the menu update that variable name directly. The second example on the form uses this technique to display the results of a call to the `GETFILE()` or `GETDIR()` function initiated by the shortcut menu called from its *RightClick()* method.

How can I format text correctly? (*Example: frmFormat.scx*)

The problem with this is that the only native Visual FoxPro function that even attempts to deal with this issue is the `PROPER()` function. However, it simply changes all text to lowercase and then forces the first letter of each "word" to uppercase. The question, then, is what constitutes a "word"? The answer appears, from experimentation, to be a *carriage return*, *space,* or *tab* character. The consequence is that the string:

```
WHICH IS BETTER FOR UPDATING TABLES? (SQL OR NATIVE FOXPRO COMMANDS)
```

is returned by the `PROPER()` function as:

```
Which Is Better For Updating Tables? (sql Or Native Foxpro Commands)
```

Which is rather less than satisfactory. The situation is even worse when we consider names; **Table 1** gives the `PROPER()` version of some common names.

***Table 1**. Using Proper() with names.*

Name	Proper(Name)
O'Brian	O'brian
MacKay	Mackay
McNair	McNair
LeFevre	Lefevre
de la Mere	De La Mere
van Beilen	Van Beilen

The problem

In fact there are two problems with writing code to deal with text formatting. First, there are really no generic rules that can be applied—especially when we have to deal with names, or technical or business language. For example, how could we write generic code to differentiate between "*Mackenzie*" and "*MacKennie*", or between an acronym like "*SONAR*" and the word "*sonic*", or between "*MR*" as the abbreviation for "*Mister*" and the abbreviation for the British High Court judge known as the "*Master of the Rolls*", or even just to recognize that the letters "*MA*" refer to someone's college degree rather than their mother?

Second, we have to deal with the issue of defining "words", and this is not as easy as it might appear. Version 7.0 has brought into the language two functions, `GETWORDCOUNT()` and `GETWORDNUM()`, which have, in previous versions, only been available in the FoxTools library (as *WordNum()* and *Words()*, respectively). These functions have the capability to accept a specific delimiter as the separator to use for determining the spacing for words, but the problem is that it is entirely possible to have more than one delimiter in a single string. For example, to parse our test string correctly, we need to recognize that both the spaces and the opening parenthesis delimit "words".

The solution

The only real solution that we have found, in the absence of generic rules, is to data drive the process of formatting so that exceptions can be handled on a case-by-case basis. This makes our rules very easy—we retrieve each word, force it to uppercase, and see whether the result exists in our table of exceptions. If so, we use the format that is defined in the table; otherwise, we simply capitalize the first letter and force the remainder to lowercase. There are still, as we shall see, some issues with words that contain apostrophes, but they are handled as part of the solution to the second part of the problem.

The second part of the problem is to determine what constitutes a word. The solution we have adopted is to force the input string into a standard format internally by replacing all existing spaces with a specific identifiable character (we use `CHR(96)`, the "`" mark). Next we parse the string and add spaces after every character that is neither a letter nor a digit. This changes our test string to:

```
WHICH` IS` BETTER` FOR` UPDATING` TABLES? ` ( SQL` OR` NATIVE` FOXPRO`
COMMANDS)
```

Once we have the string in this format we can use the standard word-based functions to retrieve the "words," which are now only delimited by spaces, and apply formatting. Finally, we restore the original spacing of the string.

The xchgcase class (*Example: WordForm.prg, WordForm.dbf*)

This class implements the solution outlined in the preceding sections. It is based on the Visual FoxPro Session base class and so uses a private datasession to open its associated table (named "wordform"). This has an added benefit in that we do not have to worry about saving and restoring the working environment (for instance, we can force **EXACT = ON** safely because it will only affect the datasession used by this class).

The class has a single exposed method, *FormatText()*, which accepts an input string as a parameter. The string is first forced to the standardized internal format by calling the *ForceSpacing()* method which populates the *cOutString* property with the re-formatted string:

```
PROTECTED FUNCTION ForceSpacing()
LOCAL lnLen, lnCnt, lcSce, lcTgt, lcChar
  *** Firstly process all spaces to CHR(96) to preserve them
  lcSce = STRTRAN( This.cInstring, CHR(32), CHR(96))
  *** Get the overall length
  lnLen = LEN( This.cInstring )
  *** Process each character and write it out so that everything
  *** except letters and numbers is followed by a space
  STORE  "" TO lcChar, lcTgt
  FOR lnCnt = 1 TO lnlen
    *** Get a Character
    lcChar = SUBSTR( lcSce, lnCnt, 1 )
    *** Is it a letter
    IF ISALPHA( lcChar )
      lcTgt = lcTgt + lcChar
      LOOP
    ENDIF
    *** Not a letter, is it a number?
    IF ISDIGIT( lcChar )
      lcTgt = lcTgt + lcChar
      LOOP
    ENDIF
    *** Neither letter nor number so add a space
    lcTgt = lcTgt + lcChar + CHR(32)
  NEXT
  This.cOutString = lcTgt
  RETURN
ENDFUNC
```

Next the *ForceCase()* method parses the resulting string using the appropriate function to retrieve each "word". The first part of the processing checks for and removes any terminating character that has been added by the process of forcing the spacing. If the resulting "word" does not contain any letters or numbers (that is, only punctuation or non-printable characters), it is ignored.

```
PROTECTED FUNCTION ForceCase()
LOCAL lnWords, lcSce, lnCnt, llAddMarker, llAddLastChar, lcWord, lnLastChar
  lnWords = This.nWordCount
  *** Get the string to work with
  lcSce = This.cOutString
  *** Clear the output property
  This.cOutString = ""
  *** Then Process each word in turn
  FOR lnCnt = 1 TO lnWords
    *** Initialize flags
    STORE .F. TO llAddMarker, llAddLastChar
    *** Use correct function
    IF This.nFoxVersion = 700
      *** Use the VFP 7.00 function
      lcWord = GETWORDNUM( lcSce, lnCnt )
    ELSE
      *** Use the FoxTools function
      lcWord = WORDNUM( lcSce, lnCnt )
    ENDIF

    *** First, strip off the space marker if we have one
    IF RIGHT( lcWord, 1) = CHR(96)
      llAddMarker = .T.
      lcWord = LEFT( lcWord, LEN(lcWord) - 1)
    ENDIF
    *** And check for any terminating punctuation marks
    lcLastChar = RIGHT( lcWord, 1 )
    IF ISALPHA( lcLastChar ) OR ISDIGIT( lcLastChar )
      *** It's either a letter or number, so do nothing
    ELSE
      *** It's something we don't want here so lose it
      llAddLastChar = .T.
      lcWord = LEFT( lcWord, LEN(lcWord) - 1)
      *** Replace non-printable characters with spaces
      lcLastChar = IIF( ASC( lcLastChar ) < 32, " ", lcLastChar )
    ENDIF
    *** If all we have left is a space - ignore it
    IF ! EMPTY( lcWord )
      *** Is it a specially formatted word?
      IF SEEK( UPPER( lcWord ), 'wordform', 'cwdupper' )
        *** Yep, just use the specified format
        lcWord = ALLTRIM( wordform.cwdformat )
        *** Ensure the first word is capitalized Whatever it is
        IF lnCnt = 1
          lcWord = UPPER( LEFT( lcWord, 1 )) ;
                 + IIF( LEN( lcWord ) > 1,  SUBSTR( lcWord, 2 ), "" )
        ENDIF
      ELSE
        *** Just force to simple PROPER() case
        lcWord = PROPER( ALLTRIM( lcWord ))
      ENDIF
    ENDIF
    This.AddToOutPut( lcWord, llAddLastChar, lcLastChar, llAddMarker )
  NEXT
ENDFUNC
```

The second part of the method is concerned with validating the word against the list of words in the table. If an exact match is found, the formatting defined in the table is applied

"as-is" unless the word happens to be the first in the string, in which case it is forced to have a leading capital letter anyway. If no match is found, the native **PROPER()** function is used to format the word.

On completion, the *RestoreSpacing()* method is called to remove the **CHR(96)** characters and replace them with spaces so that the input string is now back in its original format. The last part of the process is handled by the *CheckTerminalCaps()* method and is concerned with removing any spurious capitalization that may have occurred when the *ForceCase()* method re-formatted the string. The reason is that words like "we're" and "isn't" will be treated as two separate words and so will now look like "we'Re" and "isn'T" respectively. The essential part of this method is inside the **FOR…NEXT** loop that parses each word in the output string:

```
*** Do we have a terminal "'" in this word
lnAPos = RAT( "'", lcWord )
*** If so, is it further in than Position 2
*** (ie NOT O'xxx or d'xxx or l'xxx)
  IF lnAPos > 2
    LOCAL lnStPos, lcMakeLower, lnReplaceWith
    *** Force everything after the apostrophe to lower case
    lcMakeLower = LOWER( SUBSTR( lcWord, lnAPos + 1 ) )
    *** And re-build the word, and update the output string
    lcReplaceWith = LEFT( lcWord , lnAPos ) + lcMakeLower
    This.cOutString = STRTRAN( This.cOutString, lcWord, lcReplaceWith )
  ELSE
    IF lnAPos > 0
      *** Is the last letter capitalized?
      IF RIGHT( lcWord, 1 ) = UPPER( RIGHT( lcWord, 1 ))
        LOCAL lnStPos, lcChar, lnLetterPos
        lcChar = LOWER( RIGHT( lcWord, 1 ))
        *** Find out where in the string we are
        lnStPos = AT( lcWord, This.cOutString )
        *** And where the offending letter is
        lnLetterPos = lnStPos + LEN( lcWord ) - 1
        *** Now re-build the Output string
        This.cOutString = LEFT( This.cOutString, lnLetterPos - 1 )
                        + lcChar + SUBSTR( This.cOutString, lnLetterPos + 1 )
      ENDIF
    ENDIF
  ENDIF
```

The following code snippet shows how the class handles our original test string:

```
loFmt = NEWOBJECT( 'xChgCase', 'wordform.prg' )
lcStr = [WHICH IS BETTER FOR UPDATING TABLES? (SQL OR NATIVE FOXPRO COMMANDS)]
? loFmt.FormatText( lcStr )
```

This now results in:

```
Which is Better for Updating Tables? (SQL or Native FoxPro Commands)
```

which is a lot better than the result we got from simply applying the **PROPER()** function.

The sample form (see **Figure 7**) includes various test strings and shows how the formatter handles them and also allows you to add your own specifically formatted words to the exclusion table.

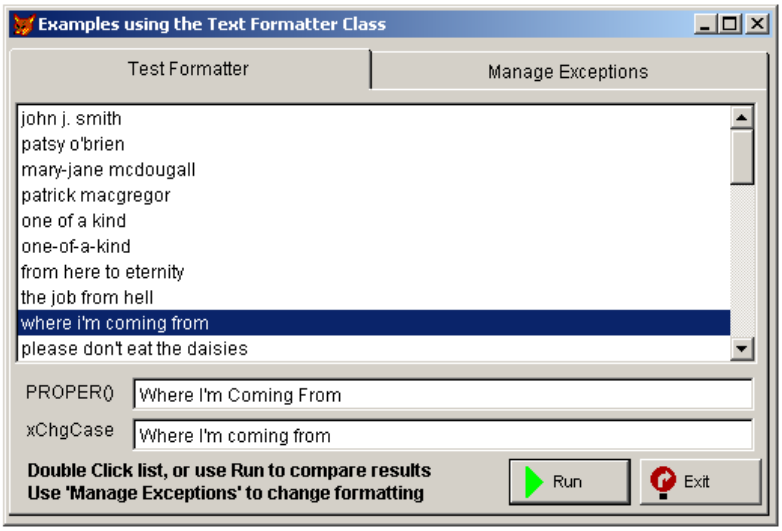


***Figure 7**. Using the text formatting class (frmformat.scx).*

How do I data drive object instantiation? (*Example: Factory.prg*)

We all know how to instantiate objects and do so every day in our applications using **`CREATEOBJECT()`**, **`ADDOBJECT()`**, or **`NEWOBJECT()`**. These commands are both simple and straightforward, so why on earth would we ever want to data drive this functionality? One very good reason is that whenever a class in an application has to be replaced by a different class, there is a problem. First we have to search the entire application to find all of the places where the original class was used. Then we must change every occurrence of the code to instantiate the new class. It is almost inevitable that we will miss at least one occurrence, or worse yet, introduce new bugs into the code by making a mistake when typing the new class name. By data driving object creation, using a *Factory* pattern, we can eliminate these problems. Instead of having to change code when circumstances change, we simply change the metadata.

A Factory pattern is defined as the provision of "*an interface for creating families of related or dependent objects without specifying their concrete classes*" (*Design Patterns: Elements of Reusable Object Oriented Software*, Gamma, Helm, Johnson, and Vlissides, Addison Wesley, 1977, ISBN 0-201-63361-2).

This sounds impressive, but what does it really mean? Quite simply, if we separate the name of the class that is to be instantiated (that is, the "concrete" class) from the process that instantiates it, we gain a lot of flexibility in determining the specific classes that provide our functionality at any given point in time. To accomplish this, we need a "Classes" table with the structure listed in the **Table 2**.

***Table 2**. Structure of Classes.dbf.*

Field name	Data type	Width	Purpose
CKey	C	20	Keyword used to uniquely identify the record.
cClassName	C	50	Name of concrete class to instantiate.
cLibrary	C	50	Name of class library (vcx or prg) that holds the class definition.
lActive	L		Active entry indicator.
Properties	M		Contains miscellaneous information in the form attribute=value, for example, to set properties of the object to specific values after it is instantiated.

In order to instantiate any of the classes listed in the table, we invoke the Factory's custom *New()* method, passing it a keyword and any parameters. For example, to instantiate the "Cut, Copy, Paste" shortcut menu discussed earlier in this chapter, this is the only code required:

```
Factory.New( 'ShortcutMenu', 'CutCopyPaste' )
```

provided that we have an entry in our classes table that looks like this:

```
cKey = 'shortcutmenu'
cClassName = 'xPopMenu'
cLibrary = 'Popmenu.prg'
```

The Factory object has three custom methods, described in **Table 3**.

***Table 3**. Factory class custom methods.*

Method	Purpose
New	Instantiates an object and returns an object reference if successful, null if not.
GetClassInfo	Called by *New()*, uses the passed keyword to find the correct record in CLASSES.DBF. This record contains the name of the class to instantiate and the name of the class library in which it is stored. Returns an object with *cClassName* and *cLibrary* properties that are populated from the record if it is found in the classes table. If the keyword is not found in the table, returns null.
ChkLibType	Called by *New()*, verifies the existence of the cLibrary in the classes table, determines if it is a vcx or a prg, and returns the appropriate extension.

If we want to instantiate a different class, or if we move the class definition to a different class library or program file, all we need to do is modify the appropriate fields in CLASSES.DBF. There is no need to modify any code. The following code, in the Factory's *New()* method, instantiates the specified object and returns an object reference. If the Factory is unable to create the required instance, it returns .NULL. So we can check for the existence of an object that is produced by our Factory in the same way that we do when we use **CREATEOBJECT()** or **ADDOBJECT()**.

```
LOCAL loClassInfo, lcLibType, lcCommand, lnParm,;
lnParmCount, loObject
```

```
*** Make sure we got passed a keyword
IF EMPTY( tcKey )
  RETURN .NULL.
ENDIF

*** Get the class information
loClassInfo = This.GetClassInfo( tcKey )

*** Make sure keyword was found in classes table
IF ISNULL( loClassInfo )
  RETURN .NULL.
ENDIF

*** Is this class in a vcx or a prg?
lcLibType = This.ChkLibType( loClassInfo.cLibrary )

IF EMPTY( lcLibType )
  RETURN .NULL.
ELSE
  loClassInfo.cLibrary = FORCEEXT( loClassInfo.cLibrary, lcLibType )
ENDIF

lcCommand = 'NewObject( "' + loClassInfo.cClassName + '", "' + ;
               loClassInfo.cLibrary + '"'

*** Now tack the parameters on to the end of the command
*** if any were passed
lnParmCount = PCOUNT() - 1
IF  lnParmCount > 0
  lcCommand = lcCommand + ', ""'
  FOR lnParm = 1 TO  lnParmCount
    lcCommand = lcCommand + ', tuParam' + TRANSFORM( lnParm )
  ENDFOR
ENDIF
lcCommand = lcCommand + ' )'

loObject = &lcCommand

RETURN loObject
```

The custom *GetClassInfo()* method uses the keyword that was passed to the Factory's *New()* method to look up the class and class library in the classes table. The following code fragment illustrates how this method has been coded to allow for the use of a hierarchical set of classes tables that can be searched in sequence.

```
*** Check to see if the developers are using a local classes table
*** to test work in progress. If there is one, use the information
*** from the local table if it is there. Only check the application
*** classes table if the keyword can't be found in the local one
lnSelect = SELECT()
IF This.lWIPTable
  SELECT WIPclasses
  LOCATE FOR UPPER( ALLTRIM( cKey ) ) == lcKey
  llFound = NOT EOF()
ENDIF
```

```
*** Now check the application classes table if
*** we need to
IF NOT llFound
  SELECT Classes
  LOCATE FOR UPPER( ALLTRIM( cKey ) ) == lcKey
  llFound = NOT EOF()
ENDIF

*** Package up class details in an object
*** to send back to the caller
IF llFound
  SCATTER NAME loClassInfo FIELDS cClassName, cLibrary
  loClassInfo.cClassName = UPPER( ALLTRIM( loClassInfo.cClassName ) )
  loClassInfo.cLibrary = UPPER( ALLTRIM( loClassInfo.cLibrary ) )
ELSE
  loClassInfo = .NULL.
ENDIF
SELECT ( lnSelect )

RETURN loClassInfo
```

Thus a developer can have a local "work in progress" classes table that is entirely separate from the application's definitive "production" classes table. By ensuring that the local table is searched for the passed keyword before the "application level" classes table, it is possible to test new functionality without having to add anything to the production tables—thereby eliminating the risk of introducing crashing bugs into production code. This is especially important when you are developing in a team environment.

How do I data drive a migration? (*Example: Migrate.prg and FieldMap.dbf*)

The need to migrate data from one structure to another is common when business requirements change and data structures no longer support the business model. Such a process is "one-off" because, eventually, the migration of data to the new structure will be complete and the migration process will never be required again. When confronted with this situation, it is very tempting to write a "quick and dirty" program that hard-codes all of the steps required to convert the data. This is a *bad move!*

Anyone who has ever written a migration program can attest to the fact that it is invariably an iterative process. After careful examination of the old data, the first attempt is made to migrate this data into the new structure. When the results are reviewed by knowledgeable end users (they are the data owners, after all), it is certain that new information will begin to surface. If you hard-code everything from the start, phrases like "*Oh, I forgot to tell you that the xyz field contains customer balances unless the data was entered on a rainy Tuesday, in which case it contains a credit memo*" will haunt your dreams. The process of refining the migration during this iterative process is much easier when the process is data driven. The objective is to change data in a table rather than code.

The key to data driving a migration is the construction of a "field map" that contains the rules for moving the legacy data into the data structures for the new system. It typically has a structure similar to the one listed in **Table 4**.

***Table 4**. Typical "field map" record structure.*

Field name	Data type	Field length	Explanation
cProcName	C	30	Used to tie sets of tables together for processing. Generally corresponds to the name of a method in the subclass that is handling the particular migration process.
iSeqNo	I		Specifies the order in which mapping records for a particular cProcName should be processed.
cSourceTbl	C	30	Name of the source table containing this datum.
cSourceFld	C	30	Name of the field in the source table.
cTargetTbl	C	30	Name of the table in which to place the datum.
cTargetFld	C	30	Name of the field in the target table.
cConstant	C	10	Contains the string equivalent of any constant values required in this field of the target table. Can be used to populate new fields with a default value.
mRuleText	M		Rule text that is evaluated to supply the value for the target field.
lCreateNew	L		When true, the migration program creates a new record in the target table.

Populating such a field mapping table can be a very tedious task. Fortunately, we can write a program that will, at least partially, automate the process. We can open the database container for the new data structures and scan it, processing the tables and fields it contains, and use this information to populate the cTargetTbl and cTargetFld fields in the field map. Alternatively, we could process the database container that holds the legacy data (if there is one), and populate the cSourceTbl and cSourceFld fields. However, since we are typically normalizing data when we migrate it into a new structure, it probably makes more sense to do the former (your mileage may vary). The following code snippet illustrates how FIELDMAP.DBF can be partially generated programmatically.

```
LOCAL lcDBC, lcTable

*** Get the path to the source data
*** and open the dbc as a table
lcDBC = "CH02"
USE ( lcDBC )

USE FieldMap IN 0

*** Now fill all the target table and target field fields
*** in the field map with the information from the new system
SCAN FOR ObjectType = 'Table'
  lcParentID = ObjectID
  lcTableName = ObjectName
  SELECT ObjectName FROM ( lcDBC ) ;
    WHERE ParentID = lcParentID AND ObjectType = 'Field' ;
      INTO CURSOR Temp
  SCAN
    INSERT INTO FieldMap ( cSourceTbl, cSourceFld ) ;
      VALUES ( lcTableName, Temp.ObjectName )
  ENDSCAN
ENDSCAN
```

It is a simple matter to write a migration program to process each set of records in the field map file that shares the same *cProcName*. The trick is to process the fields that govern the migration in order. This code is executed for each record in the field map.

```
DO CASE
  CASE NOT EMPTY( FieldMap.mRuleText )
    *** Evaluate the rule text field
    luValue = EVAL( ALLTRIM( FieldMap.mRuleText ) )

  CASE NOT EMPTY( FieldMap.cConstant )
    *** Make sure the constant has the correct data type
    *** so find out what it should be in the target table
    lcType = TYPE( ALLTRIM( FieldMap.cTargetTbl ) + ;
             '.' + ALLTRIM( FieldMap.cTargetFld ) )
    *** And Convert it because all constants are stored
    *** as character strings in the field map
    luValue = Str2Exp( ALLTRIM( FieldMap.cConstant ), lcType )

  OTHERWISE
    *** It is a field from the specified source table
    luValue = EVAL( ALLTRIM( FieldMap.cSourceTbl ) + '.' + ;
                    ALLTRIM( FieldMap.cSourceFld ) )
ENDCASE

*** And replace the specified field in the target table
REPLACE ( ALLTRIM( FieldMap.cTargetFld ) ) WITH luValue ;
  IN ( ALLTRIM( FieldMap.cTargetTbl ) )
```

If the mapping record for the current item contains some rule text, the rule is evaluated and the result is used to populate the target field. If there is no rule text, the program checks to see whether a constant value is specified. The use of a constant is very handy to have for those occasions where the target field is a brand-new field in the new system and we want to supply some default value for it in the migration (like "Migrated on *yyyy-mm-dd*", for example). If there is no rule text and there is no constant, the field specified from the legacy data is transferred "as is" to the new system.

The rule text in the field map can be something as simple as a FoxPro function that returns a formatted result (for instance, **DATETIME()**, **PADL()**, **PADR**, and so on) or a very complex transformation that is performed by either a function in your migration program or a method of your migration class. All that is required is to specify either *MyComplexFunction()* or *This.MyComplexMethod()* as the rule text in the field map, and VFP will not even complain that "This can only be used inside of a method" because the rule text *is* being evaluated inside of a method!

> ! *The sample program assumes that you have installed the samples that ship with Visual FoxPro!*

How do I data drive data validation? (*Example: Validation.vcx::Validator and ValidationDemo.scx*)

It does not matter how large, or small, your application is. Whether it runs as a stand-alone application on a desktop, or is an enterprise-wide solution running over the Internet, it will

benefit by having its validation rules data driven. It is inevitable that as businesses evolve, the rules governing the way in which they operate will change. It is always much easier to keep up with these changes if all that is required is to change a rule in a table. (A side benefit, already mentioned earlier, is that we also minimize the risk of introducing new bugs into production code.)

The first question when attempting to data drive your business rules is where to store them. If you are running a Visual FoxPro application, there is a ready made repository in the Database Container which, with the advent of Visual FoxPro Version 7, now has even more functionality with the introduction of Database Events. However, by doing so you are severely limiting the scalability of your application. A much better approach is to keep this data in a separate table that can, if needed, be converted into whatever database is needed along with the rest of the data.

The next question is how to store the rules. The approach we favor is to use a simple table (named "BIZRULES.DBF") that contains a list of table and field names together with a memo field (see **Table 5**). An important additional item here is the error number associated with a rule that provides a look-up into our standard error message table.

***Table 5**. Structure of the BizRules table.*

Field name	Data type	Field length	Explanation
cTable	C	30	Name of the table in which the field is used
cField	C	30	Name of the field to validate
mRuleText	M	4	Validation rule text
iErrorNum	I	4	Error number associated with the rule

The way in which we store the rule text is important. It must always take the form of a statement that can return a logical value when it is evaluated, indicating whether the validation succeeded or failed. Typically this will be either an `IIF()` statement or the name of a method (procedure or function) that returns a logical value. It is worth noting that since this table will be read by an object, we can refer to methods of that object directly in the *mRuleText* field because, when the field is evaluated, it will be inside an object and a reference to "*This*" will not cause Visual FoxPro to complain that "This can only be used inside of a method."

Having created our table, it is a simple matter to define a class that can be instantiated from within a form method, or by a VFP COM component, that has a single exposed *Validate()* method in its public interface. The calling object merely has to instantiate the object and pass the relevant table and field information to the *Validate()* method. The caller has no need to know anything other than whether the validation succeeded or failed, and if it failed, what error numbers are involved. This, of course, means passing back more than one piece of data, and so we use parameter objects for transferring data back and forth. (There are a couple of side benefits to using parameter objects: they allow you to use named parameters and you can pass arrays by value.)

The sample form included with this chapter saves the *ControlSources* of all the bound controls that it contains when it is instantiated. The form's custom *SaveControlSources()* method saves them to the *aFieldList* property of the form like this:

```
LPARAMETERS toControl
LOCAL loControl, loPage, loColumn, lnFieldCnt

*** Spin through all the bound controls on the form
*** and add their controlSource the form's
*** aFieldList array
DO CASE
  CASE UPPER( toControl.BaseClass ) = 'FORM'
    FOR EACH loControl IN toControl.Controls
      This.GetControlSources( loControl )
    ENDFOR

  CASE UPPER( toControl.BaseClass ) = 'PAGEFRAME'
    FOR EACH loPage IN toControl.Pages
      This.GetControlSources( loPage )
    ENDFOR

  CASE INLIST( UPPER( toControl.BaseClass ), 'PAGE', 'CONTAINER' )
    FOR EACH loControl IN toControl.Controls
      This.GetControlSources( loControl )
    ENDFOR

  CASE UPPER( toControl.BaseClass ) = 'GRID'
    FOR EACH loColumn IN toControl.Columns
      This.GetControlSources( loColumn )
    ENDFOR

  OTHERWISE
    IF PEMSTATUS( toControl, 'ControlSource', 5 )
      IF NOT EMPTY( toControl.ControlSource ) AND ;
         NOT toControl.ReadOnly
        *** How many Fields in the array currently?
        lnFieldCnt = ALEN( This.aFieldList, 1 )
        *** If only one row, is it actually a field?
        IF lnFieldCnt = 1 AND EMPTY( This.aFieldList[ 1, 1 ] )
          *** Nope - Field count = 1
          lnFieldCnt = 1
        ELSE
          lnFieldCnt = lnFieldCnt + 1
        ENDIF
        DIMENSION This.aFieldList[ lnFieldCnt ]
        *** Add this field to the list
        This.aFieldList[ lnFieldCnt ] = toControl.ControlSource
      ENDIF
    ENDIF
ENDCASE
```

When the user clicks on the Save button, the form's custom *Validate()* method packages up the contents of its *aFieldList* array and sends it to the validator's *Validate()* method.

```
LOCAL loErrorObj, loData

*** Package up the list of fields to send to the validator
loData = NEWOBJECT( 'Custom' )
IF VARTYPE( loData ) = 'O'
  loData.AddProperty( 'aFieldList[ 1 ]', '' )
  ACOPY( This.aFieldList, loData.aFieldList )
  *** and send it off to the validator
```

```
  loErrorObj = This.oValidator.Validate( loData )
  *** if we have errors, do not save
  IF loErrorObj.nErrorCount > 0
    *** Display the errors
    This.DisplayErrors( loErrorObj )
    RETURN .F.
  ENDIF
ELSE
  ASSERT .F. MESSAGE 'Unable to instantiate object to pass to validator'
  RETURN .F.
ENDIF
```

The validator then processes each field in the list, searching the BizRules table for a corresponding entry. If an entry exists for the field and there is rule text associated with it, the rule is applied to the field. If the validation fails, the validator adds the specified error to its internal errors collection so that an object containing the errors can be returned to the caller after all fields have been processed.

```
*** Spin through the fields we were passed and validate them
lnFields = ALEN( toFields.aFieldList )
FOR lnFld = 1 TO lnFields
  *** Look for this table and field in the BizRules Table
  lcKey = UPPER( PADR( JUSTSTEM( toFields.aFieldList[ lnFld ] ),;
            lnTableLen ) + ;
            PADR( JUSTEXT( toFields.aFieldList[ lnFld ] ),;
            lnFieldLen ) )
  IF SEEK( lcKey, 'BizRules', 'cTable' )
    *** Make sure we actually have a rule to evaluate
    IF NOT EMPTY( BizRules.mRuleText )
      IF EVALUATE( BizRules.mRuleText )
        *** peachy keen, if validation passes,
        *** our ruletext evaluates to true
      ELSE
        *** Oh Oh! Business rule violated...go ahead and log it
        This.LogErrors( BizRules.iErrorNum )
      ENDIF
    ENDIF
  ENDIF
ENDFOR

*** Now package up the error collection to return to the caller
loErrors = NEWOBJECT( 'Custom' )
IF VARTYPE( loErrors ) = 'O'
  loErrors.AddProperty( 'nErrorCount', This.nErrorCount )
  loErrors.AddProperty( 'aErrors[ 1 ]', '' )
  ACOPY( This.aErrors, loErrors.aErrors )
ENDIF

RETURN loErrors
```

When the calling object gets the response from the validation object, it can use the information to call on the services of a data-driven message manager to get the appropriate text. This can then be packaged up and formatted to provide feedback for the end user.

Chapter 3
IntelliSense, Inside and Out

Version 7.0 was notable for the introduction of three new tools to Visual FoxPro: IntelliSense, Database Events, and Installshield. Of these three, probably the most immediately apparent to us as developers, and the one with the greatest potential for improving our productivity, is IntelliSense. IntelliSense can be as simple, or as complex as you want it to be, but to really harness its potential you need to understand how it works. This chapter begins with the basics and then dives "under the hood".

IntelliSense in Visual FoxPro

The term "IntelliSense" refers primarily to the functionality that provides "as-you-type" assistance in the form of auto-completion of commands together with pop-up prompts for their available options and parameters. Although other Microsoft tools have long had this technology, it has been noticeably absent from Visual FoxPro—until the advent of Version 7.0. However, what has been introduced is far beyond what most of us expected and opens up a host of possibilities. The VFP version of IntelliSense is a very powerful productivity tool that, with a little thought, can change your life as a developer for the better.

What is IntelliSense?

The basic "out of the box" IntelliSense functionality in an editing window is illustrated by the following series of figures. We start by coding an **LPARAMETERS** statement (see **Figure 1**).

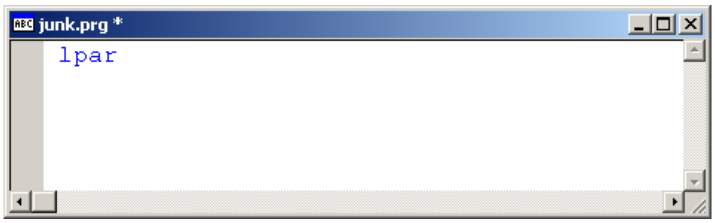


***Figure 1**. The first line of code is started…*

As usual, Visual FoxPro shows its syntax highlighting as soon as it recognizes the text as a keyword—in this case four characters is sufficient. Pressing the space bar after the text has been recognized invokes the "*Auto-Complete*" functionality (see **Figure 2**), which fills in the remainder of the command, formats it, and displays any associated "*Quick Info*" text—which looks just like a normal ToolTip.

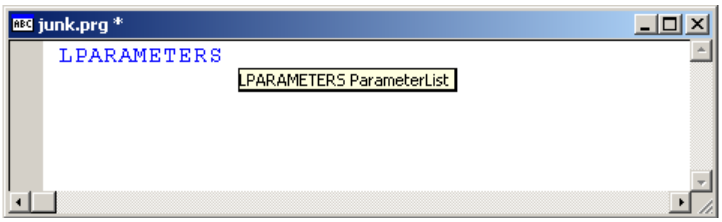


***Figure 2**. Auto-complete kicks in.*

As we move to the next line of code, which is a local variable declaration (see **Figure 3**), we see an example of the first type of IntelliSense pop-up. In this case it is another "Quick Info" item that provides an extract from the Help file for the command that we have just typed so that we can see the variants and options that are available to us at this point.

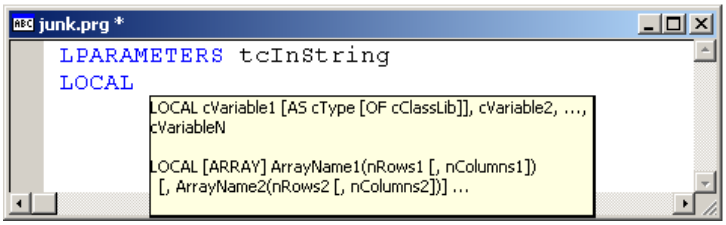


***Figure 3**. An IntelliSense Quick Info list.*

Clearly, there is no auto-complete for this command, and, by the way, notice that if we merely type the first four letters and then a space IntelliSense supplies **`LOCATE`** and not **`LOCAL`**. As always in Visual FoxPro, using the four-letter abbreviations is fine, except when the sequence in question is ambiguous, as in this case. Once we have typed **`LOCAL`** in full, there is no way that Visual FoxPro could guess at what we want to enter next but, since it does recognize the word, it can supply us with some useful information on the syntax and options for what we have entered. Having completed our declaration we now open a table with a **`USE`** command. Once we have entered the actual table name we see the second type of Quick Info list that IntelliSense offers.

This time, the list is interactive. As we scroll through the options we see the "*Designer Value Tip*" displayed for each item. Selecting an option adds the appropriate text at the current insertion point (see **Figure 4**). Notice that the same type of members list is available for the members (that is, the properties, events, and methods) of objects that:

- Are in scope in the current editing window (for example, current form and contained objects)
- Have a type library that has been opened by declaring a local variable using the **`AS`** clause (for example, **`LOCAL loWord AS word.application`**)
- Have global scope (for example, explicitly created in the command window **`goObj = CREATEOBJECT( 'myclass' )`**)

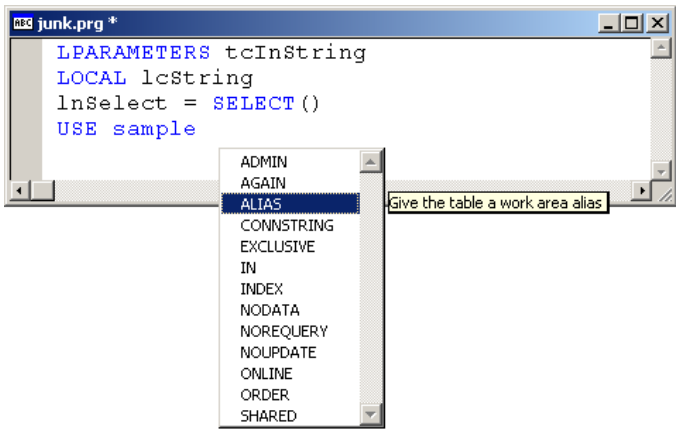


***Figure 4**. An IntelliSense Quick Info list for a FoxPro command.*

Another feature of Quick Info is illustrated in **Figure 5**. The information for the **`STRTRAN()`** function is "smart". It not only displays the list of parameters, but also tracks the insertion point as you type, highlighting the appropriate item in the parameter list. Having entered the first two parameters we are now being prompted for the third ("*cReplacement*") parameter:

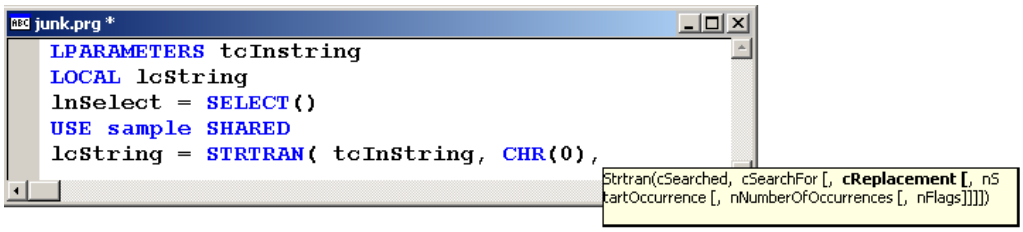


***Figure 5**. IntelliSense "smart" Quick Info.*

In addition to the various items just illustrated (which apply in any editing window), IntelliSense offers one more type of Quick Info list that is only available when you are working in the command window. This is the "*Most Recently Used*" (*MRU*) list (see **Figure 6**). Selecting an item from this list inserts the appropriate text directly into the command window.

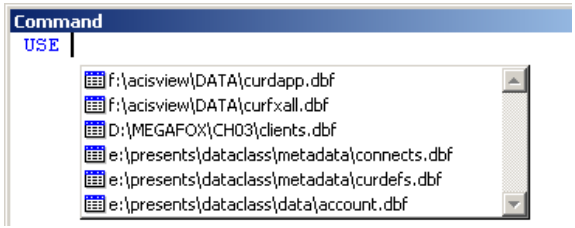


***Figure 6**. A Most Recently Used (MRU) list.*

MRU lists are available with **USE**, **MODIFY**, **OPEN**, **REPORT**, **LABEL**, and **DO** commands. Notice that the MRU list always includes the fully qualified path and file name. If the next command used was simply "**USE clients**", the entry that gets added to the list is the same as the result of using the **DBF()** function, which, in that case, would be:

```
D:\MEGAFOX\CH03\CLIENTS.DBF.
```

A variant of the MRU list is also available for the **REPLACE** command when you have a table open in currently selected work area that lists the fields in the table. If no table is open, the normal Quick Info tip for the replace command is displayed instead.

An additional feature available only in the command window is that typing "*m.*" followed by a space displays a list of all variables that are currently in scope. The value of each variable is displayed in its value tip. This feature provides a quick way of checking what is actually in memory during program execution. Just suspend the program and scroll through the list to see what it there.

For objects that have been instantiated and are in scope, or that have been explicitly declared using the new **AS** clause, another type of interactive list, the "*Members list,*" is available. This displays all of the properties, events, and methods for the object in a scrollable list. Selecting an item from the list inserts the name at the current insertion point (see **Figure 7**).

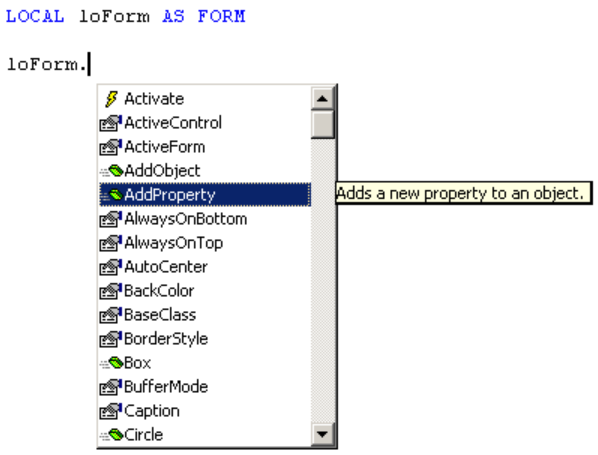


***Figure 7**. An object members list.*

This list is triggered by typing the period immediately following the object reference. Incidentally, this will not work inside a **WITH**…**ENDWITH** construct. However, by declaring a local variable (that is, **LOCAL o AS ThisForm**) you can still get the members list by typing "o.". When you have finished coding, you can use the Find and Replace tool to replace all occurrences of "o." with a plain ".".

How do I configure IntelliSense?

The Visual FoxPro Application Object (*_VFP*) exposes a new property named *EditorOptions* whose contents control the behavior of IntelliSense. The default setting is "*LQKT*" although there actually are five features that can be controlled through this property, as follows:

- *Hyperlinks* ("K" or "k")—Determines how links are activated. Setting to "k" requires only a single mouse click, while "K" requires that Ctrl-Click is used—which is the default. If neither is specified, hyperlinks are treated as normal text in editing or command windows.
- *Word Drag 'n Drop* ("W")—When enabled, text can only be dragged to a position immediately following a space. Prevents text being (inadvertently) inserted into existing text when using drag and drop in an editor or the command window. This behavior is disabled by default.
- *Designer Value Tips* ("T")—This controls whether the items in pop-up lists display any associated ToolTips. By default, they are shown as you scroll through member lists.
- *List Members* ("L" or "l")—This controls whether, and when, the object member lists are displayed. By default, lists are automatic ("L"), though you may prefer to use the "l" option to suppress the automatic display, but have the list pop up when you press Ctrl-J (or select "*List Members*" from the "Edit" pad on the main FoxPro menu).
- *Quick Info* ("Q" or "q")—This controls whether, and when, the various types of Quick Info are displayed. By default, the display is automatic ("Q"), though you may prefer to use the "q" option to suppress the automatic display, but have the Quick Info available when you press Ctrl-I (or select "*Quick Info*" from the "Edit" pad on the main FoxPro menu).

There is also a new entry under the Tools pad of the main FoxPro menu that gives you access to the "*IntelliSense Manager*" form that allows you to set the List Members and Quick Info options interactively. It also provides access to other settings that are defined in the FOXCODE.DBF table. This form, its use, and its options are well documented in the Help file under the "*Visual FoxPro IntelliSense Manager Window*" topic.

Note that "Most Recently Used" (MRU) lists do not behave in quite the way that you might expect. They only appear when you are working in the command window, and the setting of the "L" parameter in EditorOptions controls whether they appear automatically, as if they were an "Object Member List". However, because they replace the usual Quick Info display, you need to use the "Quick Info" keyboard shortcut (Ctrl-I) to display, or re-display, an MRU rather than the "Member List" shortcut (Ctrl-J).

How do I work with the FoxCode table?

This table lies at the heart of the IntelliSense functionality, and understanding how it is constructed and used is the key to making full use of the power of IntelliSense. So, at the

risk of duplicating some information that is already in the Visual FoxPro Help files, **Table 1** gives the table's structure and a brief explanation of how each field is used by the IntelliSense engine.

***Table 1**. FoxCode table structure.*

Field	Defined	Description
Type	C (1)	Identifier that defines how the record should be processed: C (Command) Auto-complete items. Triggered by " " F (Function) Quick Info items. Triggered by "(" O (COM) The Type Library to use when populating the Members List for "DEFINE AS" declarations for COM objects P (Property) Define actions when a property is accessed S (Script) Execute the script in the Data Field T (Type) The contents to use in the Members List for "DEFINE AS" declarations or for objects that do not have Type Libraries U (User) User-defined V (Version) Reserved for default/version information Z (Special) No automatic interpretation, defines custom behavior
Abbrev	C (24)	Abbreviated string to trigger the specified action
Expanded	C (26)	The string to replace the abbreviation with, where appropriate
Cmd	C (15)	The name of the script to execute for this item. Enclosed in "{}"
Tip	M (4)	The contents to display as a Quick Tip
Data	M (4)	Holds any content for this record (list values, code, script text etc).
Case	C (1)	Specifies how Expanded text is formatted to replace abbreviated text U = use Upper() function to format L = use Lower() function to format P = use Proper() function to format M or <empty> = No formatting applied X = No replacement applied Note: The value specified in the "Version" record defines the default to be used for any record that does not have its own setting.
Save	L (1)	Flag to indicate whether record is preserved during updates (False for native items)
TimeStamp	T (8)	Timestamp (VFP items only)
Source	M (4)	The source to use for record content (native items use "Reserved")
UniqueID	C (10)	Unique ID (VFP items only)
User	M (4)	Available for any user-defined information that is needed

Incidentally, one change in Version 7.0 is that, by default, the FoxCode.dbf table (along with FoxUser.dbf and the new FoxTask.dbf) is installed in the personal "user application folder" on your local drive. We are not quite sure what benefit this confers (does Microsoft really expect that several developers use the same machine and that each would want their own version of the table)? It is, however, perfectly safe to move these tables to your VFP home directory, which, in our opinion, is where they belong. If you do move these files, don't forget to change the settings in the File Locations tab of the Options dialog to reflect their new location.

The *Advanced* tab of the IntelliSense Manager form includes options to restore the FoxCode table. This means that if the table gets damaged, or even if you inadvertently delete or change some critical entry, you can simply restore the table to its original state. The nice thing about the restoration is that it will not destroy your custom items. The *TimeStamp, UniqueID,* and *Save* fields are used whenever updating or restoring the FoxCode table to determine the origin of the data and whether it may be overwritten. By default the native Visual FoxPro entries have both a time stamp and a unique ID, but their Save field is set to *False* so that they can be overwritten. User-defined entries, on the other hand, do not have either a unique ID or a time stamp, but the Save field is set to *True* so that when the table is updated, or refreshed, your user-defined items are preserved.

What are all these record types?

Each of the record types has a very specific set of functionality associated with it, and the different types indicate how the fields are interpreted by the IntelliSense engine.

Version Record (Type = "V")

There is only one of these and it is intended for internal use by Visual FoxPro. The *Expanded* field contains the version number for the current FoxCode table, and the *Case* field defines the default setting for any item that does not have one set.

Command Record (Type = "C")

This type is used for defining auto-complete text that is triggered by the space key and uses the "*Default Script*" that is defined in the *Data* field of the record with *Type* = S and an empty *Abbrev* field. All of the native commands use this methodology. However, you can also create your own "commands" that explicitly associate Quick Info (from the *Tip* field) or a Members List (from the Data field) by defining an abbreviation and including a call to the command handler script (`{cmdhandler}`) in the *Cmd* field.

To create an auto-complete command for the string "*CLD*" that will expand to "CLOSE" and display an options list offering "*Databases*" or "*Tables*" create a new record as follows:

Type	Abbrev	Expanded	Cmd	Data	Case	Save
C	CLD	Close	{cmdhandler}	Databases Tables	U	.T.

Now, typing "CLD" followed by a space in an editing window inserts the contents of the *Expanded* field ("*CLOSE*") and displays a list containing the options from the *Data* field ("*Databases*" and "*Tables*"). Selecting either adds the appropriate text. This works because both "*Close Databases*" and "*Close Tables*" are existing, expanded, entries with Quick Info tips already defined, so IntelliSense automatically displays the information from the appropriate record after completing the text.

Function Record (Type = "F")

This type is used to define auto-complete text that is triggered by the left parenthesis character "(". In this record type the contents of the Tip field are used to display the "smart" Quick Info tips that track parameter entry by matching the pattern of the text you type with that defined in the record.

To create an auto-complete entry for a user-defined function named *UseTable()*, which takes one mandatory and two optional parameters, create a new record as follows:

Type	Abbrev	Expanded	Tip	Case	Save
F	USTA	UseTable	cTableAlias[, cTag[, lExclusive]]	M	.T.

Typing "*USTA(*" now auto-completes the function name in mixed case ("*UseTable*") and displays the calling prototype with the first parameter ("*cTableAlias*") in bold. Adding a comma to the typed text moves the highlight to the second parameter, and adding another comma highlights the final option.

Property Record (Type = "P")

This type is used to assign a pop-up dialog (or value list) to be displayed whenever a value is assigned to a property whose name matches the entry in the Abbrev field. The *Cmd* field is used to indicate whether a script defined elsewhere in the table, or the contents of the *Data* field in the current record, are to be used to generate the list. The Version 7.0 FoxCode table ships with two generic scripts that are used with this record type. The first (named {*color*}) displays the color picker dialog and is associated with a number of color definition properties (for example, *BackColor*, *BorderColor*, and *FillColor*). The second (named {*picture*}) displays the open picture dialog whenever either an *Icon* or *Picture* property is assigned.

Adding the following record to your FoxCode table will display the color picker whenever a value is assigned to a property, on any object, named "*MyColor*".

Type	Abbrev	Cmd	Case	Save
P	.MyColor	{color}	M	.T.

Instead of using a pre-defined script that can be used by more than one property, you may create a script directly in the record for any property name that you want. Setting the *Cmd* field to empty braces ("{}") tells IntelliSense to use the contents of the current record's Data field. To add a script for a custom property named *cNewFile* add a record like this:

Type	Abbrev	Cmd	Data	Case	Save
P	.cNewFile	{}	LPARAMETER oFoxCode LOCAL lcTxt oFoxcode.valuetype = "V" lcTxt = ['] + GETFILE() + ['] RETURN lcTxt	M	.T.

COM Component Record (Type = "O")

This type is used to define, to the IntelliSense engine, the Type Library for a COM Component (or ActiveX control). The *Data* field is used to store the GUID (and Version) information, and the *Tip* field to store the full name of the control. The content of the *Abbrev* field is included in a drop-down list of objects associated with an **AS** clause (**DEFINE CLASS**…**AS**… , **LOCAL**…**AS**…).

The easiest way to create a record of this type is to use the IntelliSense Manager form that provides lists of registered components and controls, which can be added to or removed from the FoxCode table by checking or clearing the checkbox. However, you can insert records manually, like this.

Type	Abbrev	Tip	Data	Save
O	MSComctlLib	Microsoft TreeView Control 6.0 (SP4)	{831FDD16-0C5C-11D2-A9FC-0000F8754DA1}#2.0	.T.

Typing Record (Type = "T")

This type is used to define an entry for the drop-down list of an **AS** clause. The difference between this and the "O" record type is that there is no type library associated with this record type. Thus your own personal classes can be added to the drop-down list displayed by the **DEFINE CLASS** command. The content of the *Data* field is displayed directly in the drop-down list, and this is the only field that needs to be completed. However, we do recommend adding a description to the *Abbrev* field to make maintaining the table easier.

The easiest way to create a record of this type for visual classes is to use the IntelliSense Manager form, which allows you to select classes from your own class libraries to be added to or removed from the FoxCode table by checking or clearing the checkbox. However, for non-visual classes you have to add the records manually.

Type	Abbrev	Data	Save
T	Generic Container Class	xcntbase OF HOME()+"..\megafox\ch03\basectrl.vcx"	.T.
T	Custom Header Class	BaseHdr OF D:\MEGAFOX\NONVISCLASSES.PRG	.T.

In fact, for records defined as Type "T" IntelliSense merely inserts whatever text is included in the Data field so it can also be used to insert a line (or even multiple lines) of text or code—although since it is only triggered by an **AS** clause, we are not quite sure what value this piece of information has.

User Record (Type = "U")

This record type is used to identify abbreviations for user-defined content. It differs from the *Command* type in that it replaces the content of the *Abbrev* field with the content of the *Expanded* field. Instead of just "completing" text, it actually substitutes text—more like a keyboard macro than an auto-complete. There is no need to have the expanded text related to the abbreviation that triggers it.

Type	Abbrev	Expanded	Case	Save
U	Copyright	©Tightline Computers Inc	X	.T.

You can also associate a script with a User Type record by including empty braces ("{}") in the *Cmd* field. This indicates to the IntelliSense engine that the Data field of the record contains script code that is to be executed.

Type	Abbrev	Cmd	Data	Case	Save
U	Copyright	{}	LPARAMETER oFoxCode LOCAL lcTxt oFoxcode.valuetype = "V" lcTxt = "©Tightline Computers Inc, 2001" RETURN lcTxt	X	.T.

Script Record (Type = "S")

This type is used to store code as named scripts that can be executed by the IntelliSense engine. Other records trigger the execution of these scripts by including the name (enclosed in braces "{}") in their *Cmd* field. These records are used to create "generic" scripts that can be used by more than one entry. (Note: Any record type, with the exception of "T" and "O" records, can include a script in its *Data* field. However, in order to execute those scripts, a pair of empty braces "{}" must be inserted into the *Cmd* field.)

To create a script, all that is required is the name in the *Abbrev* field and the code in the *Data* field.

Type	Abbrev	Data	Save
S	ChooseFile	LPARAMETER oFoxCode LOCAL lcTxt oFoxcode.valuetype = "V" lcTxt = ['] + GETFILE() + ['] RETURN lcTxt	.T.

Custom Extension Record (Type = "Z")

This type was a late addition and did not make it into the documentation for the first release of Version 7.0. It identifies records that IntelliSense does not process automatically. In the first release of Version 7.0 the only such records were concerned with the way in which the default script handles custom properties and scripts. See "Modifying Default Behavior" for more details.

How do I create my own scripts?

All scripts consist, essentially, of two parts. The first is the IntelliSense-specific preamble and the second is the actual FoxPro code that generates the desired result. The IntelliSense-specific component usually consists of three elements.

The first is a parameter statement. Scripts need to be able to accept a single parameter, which is normally a reference to the FoxCode object:

```
LPARAMETERS toFoxCode
```

The second element sets the *ValueType* property of the FoxCode object. This property is used to determine how the result of running the code in the script is to be interpreted, and there are three possible values as shown in **Table 2**.

***Table 2**. Values for FoxCode.ValueType.*

Value	Result interpreted as	
V	Value:	Action depends upon the script—may be used to replace the typed text or add to it
L	List:	Displays the contents of the *FoxCode.Items* array as a list
T	Tip:	Displays the contents of the *FoxCode.ValueTip* property as a Quick Info Tip

The final task is to check the FoxCode object's *Location* property to determine what sort of editing window is active. Clearly not all actions are appropriate to all situations, and this

allows us to bypass the script unless we are in the correct editing window. The values generated for the different editing window types are listed in **Table 3**.

***Table 3**. Values for FoxCode.Location.*

Value	Type of editor
0	Command window
1	Program
8	Menu snippet
10	Code snippet
12	Stored procedure

These values, together with much other information about the currently active window, are obtained by calling the FoxTools _EdGetEnv() function. This is not functionality that is specific to IntelliSense.

The remainder of the script consists of normal FoxPro code.

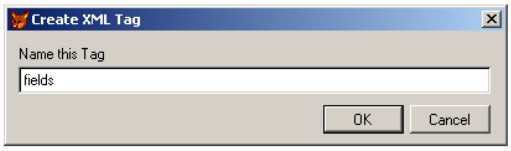
How do I create a script to insert a block of code?

In this section we will analyze a simple script that is stored directly in the *Data* field of a user-defined entry in the FOXCODE.DBF table. The record to be added looks like this:

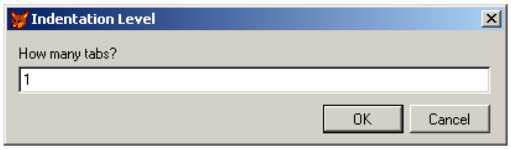
Type	Abbrev	Expanded	Cmd	Data	Save
U	xtag		{}	<Script goes here>	.T.

When the abbreviation "xtag" is typed, followed by a space, the script is executed and prompts for a name and an indentation level. The specified name is formatted as an XML tag and inserted, as illustrated at **Figure 8**.

```
*** Type the key word
xtag
*** Then a space to display the first input box
```




```
*** The second input box displays automatically
```




```
*** The result is
    <fields>
    </fields>
```

***Figure 8**. The xtag script in action.*

Setting up the script

In fact, the detailed function of the script is irrelevant. What matters is that the script creates and formats a character string that is returned for insertion at the place where the keyword was typed. The actual content of that string can be generated in any way that we wish. The script begins with a parameters statement and, because we wish the script to return a value, sets the *ValueType* property of the FoxCode object to "V":

```
LPARAMETERS toFoxCode
*** We need to return a "value"
toFoxCode.valuetype = "V"
```

For this particular example we have decided that it is not appropriate to have the "xtag" key expanded when we are working in the command window. So the next part of the script uses the *Location* property to determine the type of editing window that is active. If it is the command window, the script returns the contents of the *UserTyped* property. Effectively the script does nothing and the net effect of typing "*xtag*" in the command window is to get "*xtag*" on the current line.

```
IF toFoxCode.Location = 0
   *** Not applicable in the command window
   RETURN toFoxCode.UserTyped
ENDIF
```

Building the return string

The remainder of the script is plain FoxPro code to create, format, and return the text to insert. This code can be written using any valid FoxPro commands and functions. We have used simple string concatenation here (for ease of readability), but we could equally well have used the new **TEXT TO <variable>** to do it, providing that we first **SET TEXTMERGE = ON**.

First we declare some variables, and then we use the new (in Version 7.0) **INPUTBOX()** function to get the name of the Tag to create, and the level of indentation required. The return string is then constructed and formatted accordingly. The only unusual item here is the use of the tilde character ("~") in the line that builds the return string, immediately before the final **ENDIF**.

```
*** Here is the FoxPro Code to execute
LOCAL lcName, lcTxt, lcOpen, lcIndent, lcClose, lnLevel
STORE "" TO lcName, lcTxt, lcIndent, lcOpen, lcClose
STORE 0 TO lnLevel

*** Get the Tag Name and Indentation level
lcName = INPUTBOX( 'Name this Tag', 'Create XML Tag' )
lnLevel = VAL( INPUTBOX( 'How many tabs?','Indentation Level', "0" ))
lcName = ALLTRIM( LOWER( lcName ))
IF NOT EMPTY( lcName )
    *** Format the return string
    IF lnLevel > 0
        lcIndent = REPLICATE( CHR(9), lnLevel )
    ENDIF
    lcOpen  = lcIndent + "<" + lcName + ">"
```

```
    lcClose = lcIndent + "</" + lcName + ">"
    lcTxt = lcOpen + "~" + CHR(13) + CHR(10) + lcClose
ENDIF
RETURN lcTxt
```

Positioning the insertion point

The tilde indicates where the insertion point is to be positioned after the script has finished. (Incidentally, you can also select a block of text after insertion, by enclosing it in a pair of tilde characters.) Although the default is to use a tilde for this purpose, the actual delimiter used is determined by the *CursorLocChar* property of the FoxCode object and it can be changed to use whatever character you prefer.

Notice also that the script uses a tab character ("`CHR(9)`") to handle indentation. This is so that when the final text is inserted, the level of indentation will be modified by however the editor has been set up. In other words, it will either be left as a tab character, or be replaced by whatever number of spaces have been defined as a substitute.

There is one little snag…

Unfortunately, at the time of writing, there appears to be a minor bug with the handling of the insertion point when you have specified that tab characters be replaced by spaces. If you try this code under that condition, you will find that the insertion point is actually positioned somewhat to the left of where it should be! What is happening is that each tab at the beginning of the string is (correctly) replaced by however many spaces have been defined, but the insertion point is only being moved one character to the right for each tab. So if you define tabs as three spaces, the insertion point is wrong by two spaces for every tab inserted. At four spaces per tab, the insertion point is wrong by three spaces each, and so on.

How do I create a script to generate a list?

The IntelliSense engine handles the generation of "most recently used" and "member lists" automatically, and there is nothing more that we can do with those. The lists generated for commands and functions are actually generated from a table named FOXCODE2.DBF. A copy of this table is included with the source code, but, unlike FOXCODE.DBF, it is not exposed to developers for modification. So (unless we want to re-write the entire FoxCode application) we cannot easily alter these lists either.

However, that still leaves us with an awful lot of potential, and we can certainly define additional lists to make our own lives easier. However, dealing with lists is a little more complex than merely returning a block of code because it is actually a two-part process. First we have to generate the list, and second we have to respond to the selection that was made.

Generating the list

IntelliSense lists are created by populating an array property (named "*Items*") on the FoxCode object. This array *must* be dimensioned with two columns; the first is used to hold the text for the list items and the second to hold the Tip text to be associated with the item. In addition to specifying the content of a list, a script must tell the IntelliSense engine that a list is required. We have already seen that the *ValueType* property of the FoxCode object is used to

communicate how the result of running a script should be interpreted and, to generate a list, all that is needed is to set this property to "*L*".

The simplest way to generate a list is to create a FOXCODE.DBF record (Type = "U") in which the data field defines a script that explicitly populates the required FoxCode object properties directly, as follows:

Type	Abbrev	Expanded	Cmd	Data	Save

U	olist	lcChoice =	{}	`LPARAMETER toFoxCode`
				`WITH toFoxCode`
				`  .ValueType = "L"`
				`  DIMENSION .items[3,2]`
				`  .items[1,1] = "'First Option'"`
				`  .items[1,2] = "Tip for option 1"`
				`  .items[2,1] = "'Second Option'"`
				`  .items[2,2] = "Tip for option 2"`
				`  .items[3,1] = "'Third Option'"`
				`  .items[3,2] = "Tip for option 3"`
				`  RETURN ALLTRIM( .Expanded )`
				`ENDWITH`
					.T.

Typing "olist" followed by a space in an editing window pops up a list containing the three options defined in the script (see **Figure 9**). By returning the content of the FoxCode object's *Expanded* property, we can replace the keyword with some meaningful text and add on whatever is selected from the list.

```
*** Type the key word
olist
*** Then a space to replace the abbreviation
*** and display the options list
lcChoice =
```

```
*** The result is
lcChoice = 'Second Option'
```

***Figure 9**. A simple option list.*

While this is pretty cool, it is not really very flexible since we have to hard-code the list options directly into the script. We could create a table to hold the content of lists that we want to generate and create a separate record for each abbreviation.

Of course, we don't want to have to repeat the code that does the lookup in each record. This is where the "Script" record type comes in. As we have already seen, we can call scripts from the *Cmd* field of a FOXCODE.DBF record by including the script name in braces like this: {*scriptname*}. So we can create a generic script to handle the lookup, generate the list, and take the appropriate action when an item is selected.

*This is how the IntelliSense engine manages the lists for commands and functions using the FOXCODE2.DBF table and a different script for each **type** of list. Check out the data field for the* setsysmenu, onoffmenu, *and* dbgetmenu *script records in FOXCODE.DBF for examples.*

How to define the action when a selection is made in a list

The problem in defining the action to take when an item is selected in a list is that the code that actually creates the list is not exposed to us. So, while we can specify the content of the list, we cannot directly control the consequential action. Instead, the IntelliSense engine relies on two properties of the FoxCode object. The *Itemscript* property is used to specify a "*handler*" script that will be run after the list is closed, and the *MenuItem* property is used to store the text of the selected item. If the list is closed without any entry being selected, this property will, of course, be empty. So in order to specify how IntelliSense should respond to a selection we need to create our own handler script.

We have already seen that generic scripts require their own record (Type = "S") in FOXCODE.DBF and since they are called from another record, they must be constructed accordingly. The secret to such scripts lies in the *FoxCodeScript* class, which is defined in the IntelliSense Manager. (Note: the source code for this class can be found as FOXCODE.PRG in the FoxCode source directory.)

To create a generic script, define a subclass of the *FoxCodeScript* class to provide whatever functionality you require and instantiate it. All the necessary code is, as usual, stored in the *Data* field of the FOXCODE.DBF record. The easiest way to explain is to show it working—and it really is much easier than it sounds.

How to create a table driven list

The objective is to create a generic script that will:

- Be triggered by a simple abbreviation (the *keyword*)
- Replace the keyword with the specified expanded text
- Use that keyword to retrieve a list of items from a local table
- Display a list of the items
- Append the selected item to the expanded text

The first thing that is needed is a table. For this example we will use LISTOPTIONS.DBF, which has only two fields as shown.

Ckey	Coption
ol1	'Number One'
ol1	'Number Two'
ol1	'Number Three'
ol2	'Apples'
ol2	'Bananas'
ol2	'Cherries'

One obvious improvement to this table would be to add a column to include some tip text for our menu items, and another would be a "Sequence" column so that we can order our menus however we want. However, these refinements do not affect the principles and, to keep it simple here, we will leave such things as an exercise for the reader.

As you can see our table recognizes two keywords, "*ol1*" and "*ol2*". First we need to add a record to FOXCODE.DBF for each of these keywords. These records must define the expanded

text to replace the keyword and call the generic handler script for everything else. In this example the script is named "*lkups*" so a total of three records have to be added to FOXCODE.DBF as follows:

Type	Abbrev	Expanded	Cmd	Data	Save
U	ol1	lcChoice =	{lkups}		.T.
U	ol2	lcFruit =	{lkups}		.T.
S	lkups			<Script Code here>	

We'll describe the content of the *lkups* script in detail. The first part of the script receives a reference to the FoxCode object, instantiates the custom *ScriptHandler* object (which is defined later in the script), and passes on the reference to the FoxCode object to the handler's custom *Start()* method. (*_CodeSense* is a new VFP system variable that stores the location of the application that provides IntelliSense functionality—by default, FOXCODE.APP.)

```
LPARAMETER toFoxcode
IF FILE( _CODESENSE)
  *** The IntelliSense manager can be located
  *** Declare the local variables that we need
  LOCAL luRetVal, loHandler
  SET PROCEDURE TO (_CODESENSE ) ADDITIVE
  *** Create an instance of the custom class
  loHandler = CreateObject( "ScriptHandler" )
  *** Call Start() and pass foxcode object ref
  luRetVal  = loHandler.Start( toFoxCode )
  *** Tidy up and return result
  loHandler = NULL
  IF ATC( _CODESENSE, SET( "PROC" ) )# 0
    RELEASE PROCEDURE ( _CODESENSE )
  ENDIF
  RETURN luRetVal
ELSE
  *** Do nothing at all
ENDIF
```

This code is completely standard and you will find it repeated (with minor variations in names) in several script records. The *Start()* method in the *FoxCodeScript* base class populates a number of properties that are needed on the *FoxCodeScript* object and then calls a template method named *Main()*. This is where you place your custom code. However, the most important thing to remember is that this script will actually be called *twice*!

The first time will be when the specified keyword is typed, because the record in the FOXCODE.DBF table specifically invokes it. On this pass, the FoxCode object's *MenuItem* property will be empty—we have not yet displayed a list, and so nothing can have been selected. Therefore we must tell the IntelliSense engine that we want it to display a list. To do this we must set *FoxCode.ValueType* to "L". Next we call on the custom *GetList()* method to populate the *Items* property of the FoxCode object. Then we set *FoxCode.ItemScript* to point back to this same script so that it is called again when a selection has been made.

Finally, for this pass, we tell the IntelliSense engine to replace the keyword with the contents of the *Expanded* field.

```
DEFINE CLASS ScriptHandler as FoxCodeScript
  PROCEDURE Main()
    WITH This.oFoxCode
      IF EMPTY( .MenuItem )
        *** This is the first time this script is called,
        *** by typing in the abbreviation. First tell the
        *** IntelliSense Engine that we want a List
        .ValueType = "L"
        *** Now, pass the key to the List Builder Method
        *** This returns T when one or more items are found
        IF This.GetList( .UserTyped )
          *** We have a list, so set the ItemScript Property
          *** to re-call this script when a selection is made
          .ItemScript = 'lkups'
          *** And replace the key with the expanded text
          RETURN ALLTRIM( .Expanded )
        ELSE
          *** No items found, just return what the user typed
          RETURN .UserTyped
        ENDIF
```

You could, if you wished, make use of the *Case* field to determine how to format the return value instead of explicitly returning the contents of the *Expanded* field "as is". To do that, call the *FoxCodeScript.AdjustCase()* method in the **RETURN** statement instead; no parameters are needed. This method applies the appropriate formatting command to the content of the *Expanded* field before returning it.

The *GetList()* method is very simple indeed. It just executes a select statement into a local array. If any values are found, the *FoxCode.Items* array is sized accordingly and the values copied to it. The method returns a logical value indicating whether any items were found.

```
  PROCEDURE GetList( tcKey )
    LOCAL llRetVal
    LOCAL ARRAY laTemp[1]
    *** Get any matching records from the option list
    SELECT cOption, .F. FROM myopts ;
     WHERE cKey = tcKey ;
      INTO ARRAY laTemp
    *** Set the return value and close table
    STORE (_TALLY > 0) TO llRetVal
    USE IN myopts
    IF llRetVal
      *** Populate the foxcode ITEMS array
      DIMENSION This.oFoxCode.Items[ _TALLY, 2 ]
      ACOPY( laTemp, This.oFoxCode.Items )
    ENDIF
    RETURN llRetVal
  ENDPROC
```

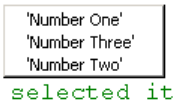
When an item is selected from the list, the script is called once more, but this time the *MenuItem* property of the FoxCode object will contain whatever was selected, which means that on the second pass the "else" condition of the *Main()* method gets executed. This now sets the *FoxCode.ValueType* property to "V" and returns whatever is contained in the *FoxCode.MenuItem* property. This benefit of this is that it gives us an opportunity to modify

the text after an item has been selected, but before it actually gets inserted. However, in this particular example we merely returned the contents of the *MenuItem* property unchanged.

```
      ELSE
        *** We have a selection so what we need to do is
        *** simply return the selected item
        .ValueType = "V"
        RETURN ALLTRIM( .MenuItem )
      ENDIF
    ENDWITH
  ENDPROC
ENDDEFINE
```

By setting the *ValueType* property to "V" we tell the IntelliSense engine to insert the return value at the current insertion point so it will appear after the expanded text (see **Figure 10**).

```
*** Type the first key word
ol1
*** Adding a space expands it
*** and displays the first list
lcChoice =
```




```
*** And the selected item is appended
lcChoice = 'Number Three'
*** The second option
ol2
*** Returns a different expansion and list
lcFruit =
```

```
lcFruit = 'Bananas'
```

***Figure 10**. Lists that share the same generic script.*

This mechanism allows us to create shortcut lists at will by simply adding the appropriate items to the LISTOPTIONS.DBF table and a single record to FOXCODE.DBF that calls our generic script.

How do I create my own Quick Info tips?

This is probably the simplest task we have tackled so far. The FOXCODE.DBF table has a field named *Tip*, which, as we have already seen can be used to provide the smart tips for either native Visual FoxPro or your own user-defined functions. The same field can also be used as the source for a Quick Info tip with other types of record.

The *FoxCodeScript* class exposes a *DisplayTip()* method that gets the tip text from the *Tip* property of the FoxCode object that it is passed. The following script shows how this is done. You will notice that the first block of code is identical to that which we used in the script to

generate a table-driven list and sets up the script object. The second block defines the custom subclass whose *Main()* method actually handles the display of the tip.

```
LPARAMETER toFoxcode
IF FILE( _CODESENSE)
  *** The IntelliSense manager can be located
  *** Declare the local variables that we need
  LOCAL luRetVal, loHandler
  *** And ensure that the base class definition
  SET PROCEDURE TO (_CODESENSE ) ADDITIVE
  *** Create an instance of the custom class
  loHandler = CreateObject( "ScriptHandler" )
  *** Call Start() and pass foxcode object ref
  luRetVal  = loHandler.Start( toFoxCode )
  *** Tidy up and return result
  loHandler = NULL
  IF ATC( _CODESENSE, SET( "PROC" ) )# 0
    RELEASE PROCEDURE ( _CODESENSE )
  ENDIF
  RETURN luRetVal
ELSE
  *** Do nothing at all
ENDIF

*** Custom Sub-Class for displaying a tip
DEFINE CLASS ScriptHandler AS FoxCodeScript
  PROCEDURE Main()
  WITH This.oFoxCode
    .ValueType = 'T'
    This.DisplayTip( .Tip )
    RETURN This.AdjustCase()
  ENDWITH
  ENDPROC
ENDDEFINE
```

Unfortunately, if you use a script in this fashion, you cannot make it do anything else, so while the ability to do it is there, we cannot immediately see a use for it. The reason is that there are only three situations in which these tips are useful. First, as Quick Info associated with items in a list, and we have seen that this is handled by the second column of the *FoxCode.Items* collection. Second, for functions, whether native to Visual FoxPro or our own, but they too are handled without needing to take this approach. Finally, for "Command" records. However, since we cannot define our own commands anyway, this is of no use unless you intend to re-define the way in which the native commands are handled. To us, this feels too much like re-inventing the wheel to be of real value.

In the absence of better information, we assume that this specific piece of functionality is exposed because it is part of the IntelliSense engine and not necessarily because it is of immediate and practical use to us as developers.

What is the Properties button in the IntelliSense Manager for?

The Advanced tab of the IntelliSense Manager has a button labeled "*Edit Properties*" that pops up a form containing (in Version 7.0) six "*custom*" properties that can be used to fine-tune the way the IntelliSense engine behaves. These properties are actually stored as *attribute* = *value* pairs in the data field of the "CustomPEMS" record in FOXCODE.DBF. Five of them are

documented in the Help file, while the sixth appears to have been a late addition. However, in our opinion, the documentation is neither precise nor complete. **Table 4** describes what we have found, in Version 7.0, that these properties actually do.

***Table 4**. IntelliSense custom properties.*

Property	Purpose
lEnableFullSetDisplay	Many **SET** commands take, as part of the basic syntax, a "**TO**" modifier. This property determines whether that "**TO**" is included as part of the IntelliSense auto-expanded command. The default value is True, but we prefer to turn this one off. The reason is that even if you type the full command yourself, IntelliSense still adds the "**TO**" with the result that you tend to get errors because you end up with a command line like this: `SET INDEX TO TO names`
lHideScriptErrors	According to the Help file this property "*Suppresses screen output of IntelliSense script errors".* The default is False but we do not detect any difference in behavior when it is set to True. If there is an error in a script, we get either the appropriate standard error message (when applicable) or a simple dialog with the text "*FoxCode Script Failure*" irrespective of the setting of this property.
lKeyWordCapitalization	Controls the auto-expansion and capitalization of the subordinate parts of commands. The default setting is True. The expansion and capitalization of abbreviations are controlled by the *Expanded* and *Case* fields in FOXCODE.DBF, but for commands that consist of more than a single word this property controls how additional words are handled.
lPropertyValueEditors	According to the Help file this property "*Enables scripts that trigger value editors for certain properties".* The default is True. In fact, this property determines whether scripts defined in FOXCODE.DBF for records with *Type = "P"* are executed. It does not affect any of the standard value lists that are defined for properties that continue to be displayed irrespective of this setting.
lExpandCOperators	Enables "*C-style*" auto-expansion of plus and minus operators: "*lcVar* ++" and "*lcVar* - -" expand as "*lcVar = lcVar + 1*" and "*lcVar = lcVar –1*" But, unlike C, VFP IntelliSense also allows us to use the "=" with any arithmetic operator to auto-expand the string. Thus: "*lcVar* +=" becomes "*lcVar = lcVar* + " and "*lcVar* *=" becomes "*lcVar = lcVar* *"
lAllowCustomDefScripts	This one must have been a late addition in Version 7.0 because it doesn't appear in the initial release of the Help file. The *FoxCodeScript.DefaultScript()* method (which is the method called whenever IntelliSense encounters a space) uses the setting of this property to determine whether a hook method named *HandleCustomScripts()* is called or not. The default is True. If you do not intend to use custom scripts to modify the default behavior, you should set this one to False because the default script is called every time you type a space in an editing window. (See "Modifying Default Behavior" for more details.)

How do I modify default behavior?

Whenever a space character is detected in an editing window, the IntelliSense engine runs the "Default Script". This script is contained in a FOXCODE.DBF script record (Type = "S") that has no abbreviation. If you examine this script you find that it determines whether it is dealing with something that it can recognize as a command and, if not, it simply exits. Next it defines

and instantiates a subclass, named *FoxCodeLoader*, of the *FoxCodeScript* class (as we showed in the scripting examples earlier in this chapter). All that it does is to define the standard *Main()* method so that it calls the *DefaultScript()* method.

```
DEFINE CLASS FoxCodeLoader AS FoxCodeScript
  PROCEDURE Main()
    THIS.DefaultScript()
  ENDPROC
ENDDEFINE
```

The first thing that the *DefaultScript()* method does is to check to see whether it has a C++ expression to expand and, if so, deals with it accordingly and exits. Next, it checks the setting of the *lAllowCustomDefScripts* property and, if True, calls the *HandleCustomScripts()* method. This method looks for, and processes, any custom scripts that have been defined *before* any more of the default behavior occurs. If a custom script returns False, further execution of the default script is prevented. This allows us to hook into the default behavior and either enhance it or provide substitute behavior by adding our own scripts.

This sequencing is, in our opinion, flawed. It seems, logically, that the code to handle C++ expansion should have been placed after *the check for custom scripts, not before it. As it stands now, you can intercept anything that triggers the default behavior* except *the expansion of C++ operators. All that can be done is to either enable or disable them entirely by setting their control property accordingly.*

In order to get a script executed as part of the default script processing, we need only do three things:

- Create the script record. As usual, the script must be able to accept a single parameter, although, in this case, it will be a reference to the script object instantiated by the default script rather than a direct reference to the FoxCode object.
- Add the name of the script, on its own line, to the *Data* field of the "*CustomDefaultScripts*" record in FOXCODE.DBF (which is a Type "*Z*" record)
- Enable the *lAllowCustomDefScripts* property. This can be accessed by clicking "*Edit Properties*" on the Advanced tab of the IntelliSense Manager. Alternatively, locate the "CustomPEMS" record in FOXCODE.DBF and edit the values directly.

You may be wondering why this is important. The answer is that the normal behavior of IntelliSense is that evaluation of what you type is only done at the beginning of a line of text. There are only two keys that will trigger IntelliSense in the middle of a line—an opening parenthesis "(" (which is used to denote a function and requires a FOXCODE.DBF record whose *Type* field is set to "F"), and the space key. By hooking into the space key handler we can have active shortcuts even while we are in the middle of a line.

For example, there are often times, when editing, that we need to embed a file name. Until now the only way to do this was either to type it directly, or copy it to the clipboard (either in Windows Explorer or by executing `_ClipText = GETFILE()` in the command window). Either

way, it was a nuisance. Now we can make our lives easier by hooking into the default script and using the built-in functionality of *FoxCodeScript* to bring up the **GETFILE()** dialog and return the selection as a character string.

First we need to create a script record and name it appropriately. The actual script is very simple. Basically we need to check the *FullLine* property of the FoxCode object for a specific string and then call the functionality that we want to implement whenever it is found. This can, of course, be anything that Visual FoxPro can legitimately execute—we are not limited to simple function calls.

In this case we use the string " GF" to trigger the **GETFILE()** dialog and wrap the return value in quotes to make it a literal. The inclusion of the leading space is necessary to prevent this from inadvertently triggered by, for example, creating a variable named "*lcLogF*".To replace the typed string with this string we use the *ReplaceWord()* function, which is one of the standard methods of the *FoxCodeScript* class. Finally, we simply return False to prevent any further action being taken in the default script.

Type	Abbrev	Data
S	InLineGetFile	LPARAMETERS toFoxCode LOCAL lcStr IF " GF" $ UPPER(toFoxCode.oFoxCode.FullLine) lcStr = ['] + GETFILE() + ['] toFoxCode.ReplaceWord(lcStr) ENDIF RETURN .F.

To activate this script just add the script name ("InLineGetFile") to the data field of the "Z" type record named "CustomDefaultScripts".

Putting IntelliSense to work

In the first part of this chapter we covered the basics of working with the IntelliSense and showed how you can use the various elements to create and modify behavior. In this section we have collected a few examples that, if you wish, you can add to your own development environment.

How do I change the behavior of browse?

One of the most irritating things that can happen when working with Visual FoxPro is that, having spent 10 minutes setting up a layout for browsing a table that has several memo fields, we inadvertently type **BROWSE** and hit the Enter key instead of typing **BROWSE LAST**. The result? We have to do it all again. Wouldn't it be nice if we could change the default behavior of the browse command so that it always uses the last configuration when we want to browse a table?

Well, we cannot actually do that, but thanks to IntelliSense we *can* tell VFP that we always want to change "**BROW**" to "**BROWSE LAST**". All that we have to do is to modify the FOXCODE.DBF record that defines the expanded form so that whenever we type **BROW** followed by a space or Enter key, it is expanded **BROWSE LAST**. This is good! However, we now have no way of executing an explicit **BROWSE NORMAL** command unless we type it in full; this is bad!

Fortunately, we can easily remedy this by adding a new record to FOXCODE.DBF that defines a "**BROWSE NORMAL**" command as the expansion for the abbreviation "*bron*". There is only one catch. If we simply copy the record for **BROWSE LAST** and edit the *Abbrev* and *Expanded* fields, we find that it doesn't work. This is because for Commands, the expanded form must actually be an expansion of the abbreviation form. Clearly "browse" is not an expansion of "bron". Of course, we could simply use a five-letter abbreviation instead, but typing "brows" plus a space doesn't really save us anything. The solution is to change the Type of the record from "Command" to "User". For user-defined records, the expanded text *replaces* the abbreviation and we are now all set. The original row (in italics) and the revised and additional rows in FOXCODE.DBF look like this:

Type	Abbrev	Expanded	Cmd	Tip	Case	Save
C	*BROW*	*Browse*	*{cmdhandler}*	*<quick info>*	*U*	*.F.*
C	BROW	Browse Last	{cmdhandler}	<quick info>	U	.T.
U	BRON	Browse Normal	{cmdhandler}	<quick info>	U	.T.

How do I insert a header into a program? (*Script: hdr*)

To insert a header, or any other block of text, we need to create a script that will return a formatted string to replace the abbreviation that triggered it. This is clearly not a generic script, so we can create it directly in the *Data* field of the FOXCODE.DBF record that defines the abbreviation like this:

Type	Abbrev	Expanded	Cmd	Tip	Data	Case	Save
U	hdr		{}	memo	Memo	M	T

The actual script, in the Data field, looks like this:

```
LPARAMETERS toFoxCode
*** If we are in the Command Window - ignore
IF toFoxcode.Location < 1
   RETURN toFoxCode.UserTyped
ENDIF
*** Return this as a value
toFoxcode.valuetype = "V"
*** Define and initialize variables
LOCAL lcTxt, lcName, lcComment, lnPos
STORE "" TO lcTxt, lcName, lcComment
#DEFINE CRLF CHR(13)+CHR(10)
lcName = WONTOP()
lcVersion = VERSION(1)
lnPos = AT( "[", lcVersion ) - 1
lcVersion = LEFT( lcVersion, lnPos )
*** Get a comment from the user
lcComment = INPUTBOX( 'Comment for the header:' )
*** Format the string
lcTxt = lcTxt + "************************************************" + CRLF
lcTxt = lcTxt + "* Program....: " + UPPER(lcName) + CRLF
lcTxt = lcTxt + "* Date.......: " + DMY(DATE()) + CRLF
lcTxt = lcTxt + "* Notice.....: Copyright (c) " ;
      + TRANSFORM( YEAR(DATE())) ;
      + " M G Akins, A Kramek & R Schummer" + CRLF
lcTxt = lcTxt + "* Compiler...: " + lcVersion + CRLF
```

```
lcTxt = lcTxt + "* Purpose....: " + lcComment + CRLF
lcTxt = lcTxt + "************************************************" + CRLF
lcTxt = lcTxt + "~"
RETURN lcTxt
```

The same pattern can be used to define any script that inserts a text string. We can either create standard "templates", similar to the header just shown, or define scripts that will avoid the necessity of repeatedly typing standard blocks of code like this:

```
WITH Thisform
  <cursor here>
ENDWITH
```

How many you define and use really depends on how many abbreviations you can comfortably remember.

How do I get a list of files? (*Script: ShoFile*)

Our first thought when this subject came up was—but we already have an automated list of "Most Recently Used files". It is configurable (on the General tab of Options dialog is a spinner for setting the number of files to hold in MRU lists), and so it can display as many entries as we want. However, we then realized that in order to get a file into the MRU list we have to use it at least once (obviously)! Furthermore unless we make the MRU list very large indeed, it really is only useful for the most *recently* used files. This is because the number of entries in the list is fixed, so once that number is reached, each new file that we open forces an existing entry out of the list. In fact, we still don't really have a good way of getting a list of all files without going through the `GETFILE()` dialog.

A little more thought gave us the idea of creating a script that would retrieve a listing of all files in the current directory, and all first-level subdirectories, of a specified type. Since we want to be able to specify the file type, we need to make this script generic and call it from several different shortcuts by adding records to the FOXCODE.DBF table as follows:

Type	Abbrev	Expanded	Cmd	Case	Save
U	mop	modify command	{shofile}	U	T
U	dop	do	{shofile}	U	T
U	mof	modify form	{shofile}	U	T
U	dof	do form	{shofile}	U	T
U	mor	modify report	{shofile}	U	T
U	dor	report form	{shofile}	U	T

As you can see, these shortcuts expand to the appropriate command to either run or modify a program, form, or report. You may add other things (for instance, classes, labels, menus, text files) as you need them. All of these call the same generic ShoFile script, which looks like this:

```
LPARAMETER oFoxcode
IF FILE(_CODESENSE)
  LOCAL eRetVal, loFoxCodeLoader
  SET PROCEDURE TO (_CODESENSE) ADDITIVE
  loFoxCodeLoader = CreateObject("FoxCodeLoader")
```

```
  eRetVal = loFoxCodeLoader.Start(m.oFoxCode)
  loFoxCodeLoader = NULL
  IF ATC(_CODESENSE,SET("PROC"))#0
    RELEASE PROCEDURE (_CODESENSE)
  ENDIF
  RETURN m.eRetVal
ENDIF
```

This block of code is the standard way of instantiating and calling a custom subclass of *FoxCodeScript* and it is used in all of the scripts that utilize that class. The second part of the script defines the custom subclass and adds the *Main()* method (which is called from *Start()*).

```
DEFINE CLASS FoxCodeLoader as FoxCodeScript
  PROCEDURE Main()
    LOCAL lcMenu, lcKey
    lcMenu = THIS.oFoxcode.MenuItem
    IF EMPTY( lcMenu )
      *** Nothing selected, so display list
      lcKey  = UPPER( THIS.oFoxcode.UserTyped )
      *** What sort of files do we want
      DO CASE
        CASE INLIST( lcKey, "MOP","DOP" )
          lcFiles = '*.prg'
        CASE INLIST( lcKey, "MOF", "DOF" )
          lcFiles = '*.scx'
        CASE INLIST( lcKey, "MOR", "DOR" )
          lcFiles = '*.frx'
        OTHERWISE
          lcFiles = ""
      ENDCASE
      *** Populate the Items Array for display
      This.GetItemList( lcFiles )
      *** Return the Expanded item
      RETURN This.AdjustCase()
    ELSE
      *** Return the Selected item
      This.oFoxCode.ValueType = "V"
      RETURN lcMenu
    ENDIF
  ENDPROC
ENDDEFINE
```

The *Main()* method merely defines the file type skeleton using the keyword that was typed and calls the custom *GetItemList()* method. This is standard FoxPro code that uses the **ADIR()** function to retrieve a list of directories and then retrieves the list of files that match the specified skeleton from the current root directory and each first-level subdirectory found. (Of course, the code could easily be modified to handle additional directory levels.) The only IntelliSense-related code in the method is right at the end where the contents of the file list array are copied to the Items collection on the FoxCode object and the *ValueType* and *ItemScript* properties are set to generate the list, and define this script as the selection handler.

```
*** If we got something, display the list
IF lnFiles > 0
  THIS.oFoxcode.ValueType = "L"
  THIS.oFoxcode.ItemScript = "ShoFile"
```

```
  *** Copy items to temporary array
  DIMENSION THIS.oFoxcode.Items[lnFiles ,2]
  ACOPY(laFiles,THIS.oFoxcode.Items)
ENDIF
```

How do I get a list of variables? (*Script: InLineGetLocVars*)

One of the items we covered in Chapter 1 ("KiloFox Revisited") was how to use the FOXTOOLS.FLL editing functions to check, and declare if necessary, a local variable in any program, procedure, or method. We thought that a useful extension would be to devise a generic script that would create a list of local variables (of the specified type) that have already been declared in the current program, procedure, or method (see **Figure 11**). This script is triggered, when needed, by typing the desired prefix followed by a space. The variable that you select is inserted and replaces the key. Just think, no more errors because of misspelled variable names!

```
*** Get the contents of the editing window into an array
*** beware! if you have more than 65,000 lines of code,
*** this will crash
lcProgText = _EdGetStr( lnHandle, 0, laEnv[ 2 ] - 1 )
lnLines = ALINES( laLines, lcProgText )

IF lnLines = ln
```

***Figure 11**. The declared variable name list.*

To be useful, such a script must be active at any point in the editing window, not merely at the beginning of a line. As we have already seen, in order to do this we have to hook into the default script, which means that the script is called every time that we type a space. Therefore the first thing we have to do in the script is to check the type of window in which we are working and that what we have typed is relevant. Having established that we are not in the command window, we check the list of prefixes that we have defined and exit immediately if we do not find what has been typed in that list.

The example assumes that you are using the "standard" prefixes of "l" plus a type identifier for Local variables (for example, lcStr, ldToday), and "t" plus a type identifier for Parameters (for example, tnValue, tlChoice). It also requires that you define local variables and parameters as comma delimited lists, and do not use continuation characters to make a single declaration statement span multiple lines of text. However, it makes no assumptions about how you actually name variables or parameters; it simply tries to match what you typed to what is in the declaration statements.

```
LPARAMETER toDefScript
LOCAL lcKey, loFoxCode

*** Get Local Ref to FoxCode Object
loFoxCode = toDefScript.oFoxCode

*** Don't want this in the command window
IF loFoxCode.Location < 1
  RETURN
ENDIF

*** Check the list of prefixes that we want to use
lcKey = LOWER( loFoxCode.UserTyped )
IF NOT INLIST( lcKey , "lc", "ln", "ll", "lo", "lu", "ld", "lt", ;
                       "tc", "tn", "tl", "to", "tu", "td", "tt" )
  RETURN
ENDIF
```

First the script retrieves all of the text in the current editing window into an array (so it is limited to a maximum of 65,000 lines of code). It then searches backwards through the text, starting at the current line number, accumulating local variable and parameter definitions as it does so. If an explicit Procedure or Function declaration is found, it stops there; otherwise, it continues to the beginning of the text.

Having built an array of all declarations, the next step is to scan it and extract and sort those that match the currently required prefix. If any are found the FoxCode object is then set up to generate a list, by setting the *ValueType* property to "L". We also set the *ItemScript* property to the name of the script that will handle the replacement of the keyword with the selected item. Finally, we copy the names we have found to the *Items* array and exit.

```
*** If we found a declaration
IF lnItem > 0
  *** Force the ValueType to designate a List
  loFoxCode.ValueType = "L"
  *** And define the Text Replacement Script as the handler
  loFoxCode.ItemScript = "ReplText"
  *** Finally copy found values to the FoxCode.Items array
  DIMENSION loFoxCode.Items[lnItem ,2]
  ACOPY( laItemList, loFoxCode.Items )
ENDIF
```

The default behavior when using lists is, as we have already seen, to replace the keyword with the expanded form and to insert the selected item immediately after it. In this case, we do not want that behavior; we merely wish to replace the trigger with whatever was selected. If we tried to do that by using the current script as the selection handler, we would find that we would be left with a single space in front of the selected item.

The solution is to use a generic script (named *ReplText*) that simply replaces whatever was originally typed with whatever was selected from a list. Both of these values are held, as properties, by the FoxCode object, and so all that this script has to do is to create an instance of the *FoxCodeScript* class and call its *ReplaceWord()* method.

```
LPARAMETER oFoxcode
IF EMPTY( ALLTRIM(oFoxcode.menuitem) )
```

```
  RETURN
ENDIF

IF FILE(_CODESENSE)
  LOCAL eRetVal, loFoxCodeLoader
  SET PROCEDURE TO (_CODESENSE) ADDITIVE
  loFoxCodeLoader = CreateObject("FoxCodeLoader")
  eRetVal = loFoxCodeLoader.Start(m.oFoxCode)
  loFoxCodeLoader = NULL
  IF ATC(_CODESENSE,SET("PROC"))#0
    RELEASE PROCEDURE (_CODESENSE)
  ENDIF
  RETURN m.eRetVal
ENDIF

DEFINE CLASS FoxCodeLoader as FoxCodeScript
  PROCEDURE Main
    LOCAL lcItem
    lcItem = ALLTRIM( This.oFoxCode.MenuItem )
    This.ReplaceWord(lcItem)
  ENDPROC
ENDDEFINE
```

Don't forget that in order to hook into the default script you must enable the *lAllowCustomDefScripts* property. This can be accessed by clicking "*Edit Properties*" on the Advanced tab of the IntelliSense Manager. Alternatively, locate the "CustomPEMS" record in FOXCODE.DBF and edit the value directly.

How do I get a list of all my custom shortcuts? (*Script: LCut*)

One minor problem that we have discovered when using IntelliSense is that it can sometimes be difficult to remember what custom shortcuts we have created. The solution, of course, is to get IntelliSense to display, on demand, a list of our shortcuts for us, and that is precisely what the *LCut* script does. This is another example of a script that generates a list, and replaces the typed value with whatever was selected, so it is very similar to the two preceding examples (see **Figure 12**).

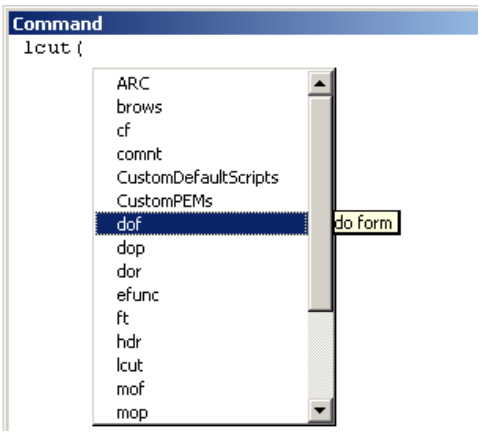


***Figure 12**. List of available shortcuts.*

The record to be added looks like this:

Type	Abbrev	Expanded	Cmd	Data	Case	Save
F	lcut		{}	<script here>		T

The only real difficulty is how to identify the things that we want to see in our list. For the purposes of this example we are using the combination of '`save = .T.`' and '`type # "S"`' as the criteria for inclusion. This excludes any of the default FoxPro entries (where *save* is always set to False), and all Script records (*type* = "S"), leaving us with only our user-defined records (this list does still include items of type "T", which are of limited use in this context). Obviously you can choose any appropriate criteria (for instance, just entries where *type* = "U") depending on how you have defined your custom shortcuts. The *GetList()* method of the *ScriptHandler* object begins by selecting data from FOXCODE.DBF:

```
PROCEDURE GetList( tcKey )
  LOCAL llRetVal, lnCnt
  LOCAL ARRAY laTemp[1]
  *** Get any matching records from the option list
  SELECT abbrev, Expanded, data, PADR(ALLTRIM(user),60);
    FROM foxcode ;
   WHERE save = .T. ;
     AND type <> "S" ;
    INTO ARRAY laTemp
  *** Set the return value
  STORE (_TALLY > 0) TO llRetVal
```

Assuming something is found, the next part of the script builds the list that will display the shortcuts. The list item tip is populated with whatever can be found by checking, in this order, the *Expanded* field, the *Data* field, and finally the *User* field. If nothing is found, the abbreviation is simply repeated. (A better solution may be to either enter a description into the *User* field or add a description column to the FOXCODE.DBF table.)

```
  IF llRetVal
    *** Populate the foxcode ITEMS array
    DIMENSION This.oFoxCode.Items[ _TALLY, 2 ]
    FOR lnCnt = 1 TO _TALLY
      This.oFoxCode.Items[ lnCnt , 1] = ALLTRIM( laTemp[ lnCnt , 1] )
      *** If we have an expanded form, use it
      IF ! EMPTY( laTemp[ lnCnt, 2] )
        This.oFoxCode.Items[ lnCnt , 2] = laTemp[ lnCnt , 2]
        LOOP
      ENDIF
      *** If we have Data, use it,
      IF ! EMPTY( laTemp[ lnCnt, 3] )
        This.oFoxCode.Items[ lnCnt , 2] = laTemp[ lnCnt , 3]
        LOOP
      ENDIF
      *** Try the USer field if nothing else
      IF ! EMPTY( laTemp[ lnCnt, 4] )
        This.oFoxCode.Items[ lnCnt , 2] = laTemp[ lnCnt , 4]
      ELSE
```

```
        *** We have nothing for this one, use the abbreviation
        This.oFoxCode.Items[ lnCnt , 2] = ALLTRIM( laTemp[ lnCnt , 1] )
      ENDIF
    NEXT
  ENDIF
  RETURN llRetVal
```

The main body of the script is identical to the script to get a list of local variables, described in the preceding section, so it will not surprise you to learn that selecting a shortcut replaces the text that invoked the list with the selected item. Notice, however, that whereas we used the default script handler to allow a space to trigger building the list of variables, in this case we have defined the script as a function. This means that to initiate it we must use an opening parenthesis rather than a space, thus: "`lcut(`"

The reason for this is so that when an abbreviation is selected by using the spacebar, it can be used to trigger its own shortcut automatically. If we had allowed the spacebar to trigger the list, the result would be the addition of an extra space to the replacement text. In other words, *two* trailing spaces would be inserted and so it would not fire the shortcut. By using a function to invoke the list we avoid this problem and can make the selected shortcut "auto-execute".

Isn't there an easier way to create a script? (*Example: is_customizer.prg*)

The short answer is that is it depends what you want the script to do. Trevor Hancock, of Microsoft, wrote a program to capture a block of selected text from an editing window and create a FOXCODE.DBF entry to insert that block of text. His program pops up a little form that allows you to define how the script record is to be created (see **Figure 13**).

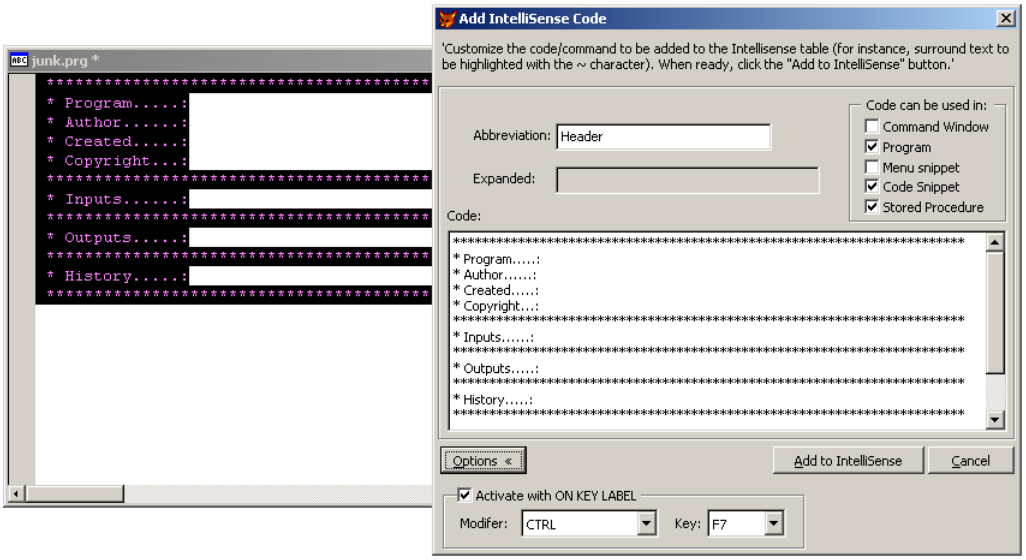


***Figure 13**. Trevor Hancock's script creation utility.*

To use this great little tool, just assign a hot key (or menu item) to call the program, and then select the text that you want to include in your script and invoke the program. We usually use a simple hot key assignment:

```
ON KEY LABEL ALT+F7 DO is_customizer
```

That is all there is to it. This is an ideal tool for quickly creating shortcuts for program headers, standard subroutines and procedures, or any other block of text or code that you use often. A very nice piece of work; thank you so much, Trevor.

Conclusion

This chapter has shown how the implementation of IntelliSense introduced in Visual FoxPro Version 7.0 goes far beyond providing the simple auto-expansion of keywords and "as-you-type" Help. It is a powerful and flexible tool that we, as developers, can use to customize, extend and enhance the native functionality of the Visual FoxPro development environment. Hopefully the examples in this chapter will help you to get to grips with this very exciting new tool and will inspire you to find new ways of using it in your daily work.

Most of the examples in this chapter are implemented by adding records to FOXCODE.DBF and so, rather than a set of programs, there is a free table named MFCODE.DBF that contains the necessary additional records. The contents of this table can simply be appended to your local FOXCODE.DBF table and will not alter your existing IntelliSense behavior in any way.

The examples that modify the behavior of Browse, and those that hook into the default script, require modifications to be made to existing records in your local copy of the FOXCODE.DBF table. These modifications will not be made automatically and you will need to make them yourself if you wish to run these particular examples.

Chapter 4
Sending and Receiving E-mail

There are many reasons that we might want to send and receive e-mail from within our applications. Perhaps we need to automate the process of sending an acknowledgement for orders received (whether placed online or entered manually). Another common use is to include e-mail as part of an error handler so that we can send detailed information about an error to developers and expedite the debugging process. This chapter shows how to implement this sort of functionality.

What are the options?

If we can be certain that Outlook is available on our user's system, then the most obvious choice is to use Outlook Automation. The Outlook object model provides a rich interface that can be easily accessed using Visual FoxPro. But not everyone has Outlook, or even uses it if they do. So what can we use besides Outlook Automation?

There are three basic options from which we can choose. First, we can use Microsoft's Messaging Application Program Interface, better known as MAPI, to send and receive e-mail from any machine on which a MAPI-compliant e-mail client is installed. Note that not all e-mail clients are MAPI-compliant, but most of them are. Outlook, Outlook Express, Groupwise, and Eudora are all examples of MAPI-compliant e-mail clients. On the other hand, Lotus CC:Mail is not.

Second, we can use Collaboration Data Objects for Windows 2000 to send and receive messages using the SMTP protocol.

Third, we could opt for a third-party tool that provides a simple interface to e-mail handling. There are many such tools on the market, but for ease of integration with VFP we would suggest looking first at the shareware "*wwipstuff*" classes which provide, among other things, support for the SMTP protocol. (For more information see the West-Wind Technologies Web site at www.west-wind.com.)

What is all this alphabet soup, anyway?

It seems to be a peculiarity of the computer industry that everything must have an obscure name and, if possible, an acronym. E-mail is certainly no exception; a brief look at the available documentation reveals a host of names and acronyms that very quickly becomes very confusing. We found "Simple MAPI", "Extended MAPI", "OLE Messaging", "Active Messaging", "CDO", "CDONTS", and "SMTP", to name but a few. What are they, and do we need to know?

What is MAPI?

MAPI is a protocol-independent architecture that separates the programming interface used by client applications from the transport mechanisms used by back-end messaging services. The acronym is derived from "Messaging Application Program Interface". It is implemented as a set of functions that can be used to add messaging functionality to Microsoft Windows based applications. The term "*Extended MAPI*" refers to the full function library and gives the

developer complete control over the messaging system on the client computer. "*Simple MAPI*" is a subset of Extended MAPI. It is limited to 12 functions that provide the ability to send and receive messages. The Microsoft MAPI Control library (MSMAPI32.OCX) that ships with Visual FoxPro is an implementation of Simple MAPI.

One important limitation is that MAPI, whether Simple or Extended, supports only plain text messages. If you want, or need, support for HTML in e-mail then you cannot use MAPI. Note: It is important to distinguish between MAPI, which is an application used to access the e-mail client, and SMTP, which is a protocol used to send e-mail over the Internet.

In order to use MAPI, a MAPI-compliant e-mail client must be installed on the local machine and MAPI simply uses whatever application is defined as the default. To find out which client is set as the default (in situations where multiple e-mail clients are installed), just open Internet Explorer and select Internet Options under the Tools menu. The Programs tab tells you which e-mail client is set as the default.

One of the consequences of this approach is that sending a message through MAPI automatically posts a copy to the Sent Items folder, just as if the message had been sent interactively. This does not happen when you access messaging services directly.

What is CDO?

Collaboration Data Objects (CDO) is, at the time of writing, the current name for what was originally called "*OLE Messaging*" and later re-named to "*Active Messaging*". (It's not surprising that people get confused, is it?) CDO comes in three forms in two versions:

- Version 1.2 exists in two forms. The first form, implemented in CDO.DLL, provides MAPI-based functionality and so is limited to plain text messages. It does not actually implement the entire functionality of Extended MAPI, but provides greater functionality than Simple MAPI. The second form, implemented in CDONTS.DLL, is SMTP-based and allows messages to contain HTML.
- CDO Version 2.0 (also known as "*CDOSYS*") provides an object model for the development of messaging applications under Windows 2000. It is based on the Simple Mail Transfer Protocol (SMTP) and Network News Transfer Protocol (NNTP) standards and is available as a system component on Microsoft Windows 2000 Server installations. (There is also a special version of CDO Version 2.0, which is only installed with Microsoft Exchange Server 2000, and is known as "*CDOEX*".)

What is SMTP?

Simple Mail Transfer Protocol (SMTP) is a core Internet Protocol used to transfer e-mail between the originator and the recipient. This protocol uses the structure of the e-mail address to determine whether the components of the message (subject line, content, attachments, and so on) can be delivered. The process is initiated when the originator's e-mail application posts a message to its designated outgoing SMTP server.

The server extracts the domain name of the recipient's e-mail address (this is the part of the address after the "@") and uses it to establish communication with the Domain Name Server (DNS). The DNS then looks up, and returns, the host name of *its* designated incoming SMTP mail server.

If all of this works properly, the originating server then establishes a direct connection to the receiving server, using Transmission Control Protocol/Internet Protocol (TCP/IP) Port 25. The originating server passes the user name (the part of the e-mail address *before* the "@") to the receiving server. If that name matches one of the receiving server's authorized user accounts, the e-mail message is transferred to await the recipient collecting their mail through whatever client program they are using.

Gotcha!

As of the time of writing, you must use either CDOSYS.DLL or CDOEX.DLL to send and receive e-mail programmatically without any user intervention if you are using Office XP or Office 2000 SP2. The Outlook security patch causes an annoying message box (see **Figure 1**) to pop up when you access the e-mail client if you are using either Simple MAPI or Outlook Automation.

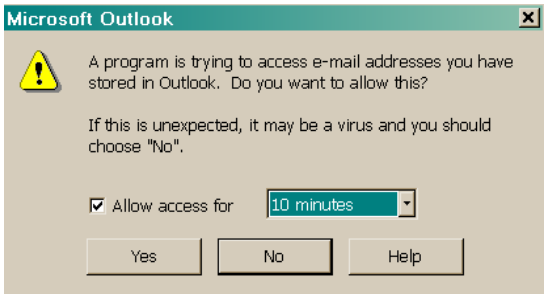


***Figure 1**. Annoying message box.*

There are a couple of ways to get around the security patch. The first is to download "Outlook Redemption", a DLL written by Dmitry Streblechenko, a Microsoft Outlook MVP, which implements the Extended MAPI interfaces to Outlook. It is available for download at www.dimastr.com/redemption/download.htm and is free unless you are distributing it in commercial software, in which case it costs $199.99. The second is to download "Express Click Yes" from www.express-soft.com/mailmate/clickyes.html.

How do I use MAPI?

To send, or receive, e-mail using MAPI, you need to instantiate two objects that are found in MSMAPI32.OCX. The *MAPISession* object is responsible for managing the mail session and the *MAPIMessages* object is used to send and receive messages. The properties, events, and methods of these two objects are quite well documented in the MAPI98.CHM Help file. If you really need some out of the ordinary implementation, you could probably get the necessary information by studying the Help file. Fortunately, for basic e-mail, you don't have to bother.

We have created a container class called *cntMapi* that that hides the complexity and makes it easy to send and receive e-mail. This class was created in the visual class designer so that it can be dropped onto a form. The class could just as easily have been built as a program file, but then would have to be instantiated explicitly in code using either **`CREATEOBJECT()`** or **`ADDOBJECT()`**.

The first thing that you need to do when setting up to send or receive e-mail using MAPI is to make sure that MAPI is installed on the local machine. This code, in the custom *IsMapiRegistered()* method of our *cntMapi* class, is called from the sample form's *Init()* and checks the Registry to ensure that MAPI is actually there. First we need to declare the constants for the Registry root keys and set up two Windows API functions:

```
*** Registry roots
#DEFINE HKEY_CLASSES_ROOT     -2147483648  && BITSET(0,31)
#DEFINE HKEY_CURRENT_USER     -2147483647  && BITSET(0,31)+1
#DEFINE HKEY_LOCAL_MACHINE    -2147483646  && BITSET(0,31)+2
#DEFINE HKEY_USERS            -2147483645  && BITSET(0,31)+3

LOCAL lnHandle, lnDataSize, lcValue, lnRes, lcSubKey, llRetVal

DECLARE INTEGER RegOpenKey IN Win32API ;
  INTEGER nHKey, ;
  STRING cSubKey, ;
  INTEGER @nResult

DECLARE INTEGER RegQueryValueEx IN Win32API ;
  INTEGER nHKey, ;
  STRING cValue, ;
  INTEGER nReserved, ;
  INTEGER nType, ;
  STRING @cBuffer, ;
  INTEGER @nSize
```

The value we are interested in is stored in the "*Software\Microsoft\Windows Messaging Subsystem*" subkey, so we first open the key, and then read the value:

```
lnHandle = 0
lnDataSize = 254
lcValue = SPACE( 254 )
lcSubKey = "Software\Microsoft\Windows Messaging Subsystem"
lnRes = RegOpenKey( HKEY_LOCAL_MACHINE, lcSubKey, @lnHandle )
IF lnRes = 0
  *** See if MAPI32.dll is there
  RegQueryValueEx( lnHandle, "CMCDLLNAME32", 0, 0, @lcValue, @lnDataSize )
  IF 'MAPI32.DLL' $ UPPER( ALLTRIM( lcValue ) )
    llRetVal = .T.
  ENDIF
ENDIF

RETURN llRetVal
```

Having determined that MAPI is registered, the next thing that you need to do is to log on. The Session object exposes a single *SignOn()* method that gets its values from four properties that must be populated prior to invoking the method.

- *UserName* This is only required if a specific user profile is being used (the default profile does not require a user name).
- *Password* This is only required if a specific user profile is being used (the default profile does not require a password).

- *DownloadMail* This specifies whether or not you want to get new mail from the host. The documentation states that if this property is set to true, new mail will be retrieved when you log on. However, this does not work as advertised when the default e-mail client is Outlook 2000 or later. It does work when Outlook Express is the default client.
- *NewSession* This specifies whether to create a new MAPI session or to use the existing session if one already exists.

If the *SignOn()* method is successful, it sets the *SessionID* property of the Session object. Now all that is left is to set the *SessionID* of the Messages object to the *SessionID* of the Session object and you are ready to send or read e-mail.

How do I read mail using MAPI? (*Example: MapiMail.scx and CH04.vcx::cntMapi*)

You may be wondering why you would ever need to *read* e-mail programmatically using MAPI, but there is a use case for it. We once worked on an application that received small downloads, on a daily basis, in the form of e-mail attachments. We needed to programmatically retrieve all unread e-mail from a specific originator that had a specific subject line, save the attached files for processing, and delete the original mail. MAPI is ideal for this sort of thing and, by hiding the complexity of managing the MAPI message and MAPI Session objects in our MAPI container class, we were able to provide an easy implementation for the client.

Before we discuss how our custom class works, let's take a brief look at the MAPI Messages object. In order to retrieve messages from the inbox of the e-mail client, you set three properties to determine which messages to retrieve, and the order in which to return them, and then call the *Fetch()* method. The properties are:

- *FetchUnreadOnly* Specifies whether only messages that have not been marked as "read" are retrieved. The default is true.
- *FetchMsgType* Specifies the type of message to retrieve. Available types are determined by the underlying mail system. The default is interpersonal message type (IPM).
- *FetchSorted* Supposedly specifies the order in which messages are retrieved, but this is not the case in VFP. Setting *FetchSorted* to either true or false makes no difference. The messages are always retrieved in the order in which they are received; in other words, the oldest message appears first.

The MAPI Messages object has a set of properties that it automatically updates with the details of the "current" message. In order to access a specific message, make it current by setting the *MsgIndex* property of the MAPI Messages object. The *MsgIndex* is a positional value that roughly corresponds to the position of the "current" message in the e-mail client's inbox. Since the *MsgIndex* is zero-based, its values can range from -1, signifying an outgoing message, to one less than the number of messages.

The properties of the currently indexed message include:

- *MsgDateReceived* The date and time, as a character string in YYYY/MM/DD HH:MM format, that the current message was received.
- *MsgNoteText* The body text of the e-mail message.
- *MsgOrigAddress* The address of the sender of the message. The messaging system sets this property for you automatically when sending a message.
- *MsgOrigDisplayName* The display name of the sender of the message. The messaging system sets this property for you automatically.
- *MsgRead* A Boolean value that indicates whether or not the current message has been read.
- *MsgSubject* The subject line of the current message. The maximum allowable length of the subject line is 64 characters.

When the current message has attachments, the *AttachmentCount* property of the MAPI Messages object is greater than zero. Accessing the attachments, if there are any, in the current message is very similar to accessing the current message itself. The MAPI Messages object has an *AttachmentIndex* property that is used to reference the current attachment. This property, like the *MsgIndex*, is zero-based, so setting the *AttachmentIndex* of the messages object to 0 allows you to access the first attachment. Once an attachment is made current, the following properties of the MAPI Messages object provide information about the attachment:

- *AttachmentName* The name of the current attachment. This is what the recipients of the e-mail see. If it is not explicitly set, the file name from the *AttachmentPathName* property is used.
- *AttachmentPathName* The fully qualified path name of the current attachment.
- *AttachmentPosition* The position of the current attachment in the message body. This is important only when sending e-mail to ensure that attachments are positioned properly.
- *AttachmentType* The type of the current attachment. Possible values are 0-Data File, 1-Embedded OLE object, and 2-Static OLE object.

How does the custom *cntMAPI* class simplify reading e-mail?

Using our custom MAPI container class makes reading e-mail a snap. All that is required is to instantiate it and pass an instance of the custom *MAPIReadParms* parameter object to its *ReadMail()* method. This parameter object, discussed shortly in detail, determines which messages will be retrieved by the *ReadMail()* method. Besides hiding the complexity of working with MAPI directly, our custom class adds functionality, enabling you to filter messages based on date received, sender, and subject line.

The class has two custom properties that are used when reading e-mail:

- *aMsgNumbers* An array of messages numbers retrieved from the e-mail client.
- *nCurrentMsg* The index into the array.

Individual custom methods access the various parts of an e-mail message (see **Table 1**). Unless otherwise stated, each method assumes that the current message is the target.

***Table 1**. Custom cntMapi methods used to retrieve message information.*

Method name	Description
DeleteMsg	Deletes the current message and re-numbers the contents of the *aMsgNumbers* collection because deleting a message re-numbers all the subsequent *MsgIndexes*.
GetAttachmentCount	Returns the number of attachments for the current message.
GetAttachmentFile	When passed the number of an attachment, returns the fully qualified file name of the attachment file from the current message.
GetBodyText	Returns the text from the message portion of the current message.
GetDateReceived	Returns the date the current message was received.
GetFirstMsg	Sets the first message in the *aMsgNumbers* array as the current message.
GetLastMsg	Sets the last message in the *aMsgNumbers* array as the current message.
GetMsg	When passed an index into the *aMsgNumbers* array, sets the message pointed to by that element as the current message.
GetNextMsg	Makes the next message in the array the current message.
GetPriorMsg	Makes the previous message in the array the current message.
GetSender	Returns the sender's e-mail address for the current message.
GetSubject	Returns the subject line for the current message.

The job of actually retrieving mail from the inbox is handled by the *ReadMail ()* method that expects to receive a single parameter object. The properties of the parameter object are passed as individual parameters to methods of the *oMapi* object. While the parameter object itself is required, all of its properties can be left at their default values if no special processing or filters are needed. The properties are described in **Table 2**.

***Table 2**. Properties of the MAPI read mail parameter object.*

Property	Description
cPassword	User password associated with the MAPI client.
cSender	When not empty, retrieves only messages from this sender.
cSubject	When not empty, retrieves only messages with this subject line.
cUserName	User name associated with the MAPI client.
dFromDate	When not empty, retrieves only messages received on or after the specified date.
dToDate	When not empty, retrieves only messages received on or before the specified date.
lDownload	When true, new mail is downloaded before processing unless the default e-mail client is Outlook 2000 or later.
lUnreadOnly	When true, retrieves only unread messages. Note that when a message has been read using MAPI (that is, *MapiMessages.MsgIndex* has been set to point at that message), the message is marked as read in the client's inbox.

The example form (see **Figure 2**) uses an instance of the *cntMapi* class (named "*oMAPI*") and an instance of the *MapiReadParms* class ("*oReadParms*") to define the parameters for filtering incoming mail. The textboxes and checkboxes on the form are bound to the properties of this object to simplify the task of populating them. So when the "Get E-Mail Now" button is clicked, all we have to do is call the form's custom *ReadMail()* method.

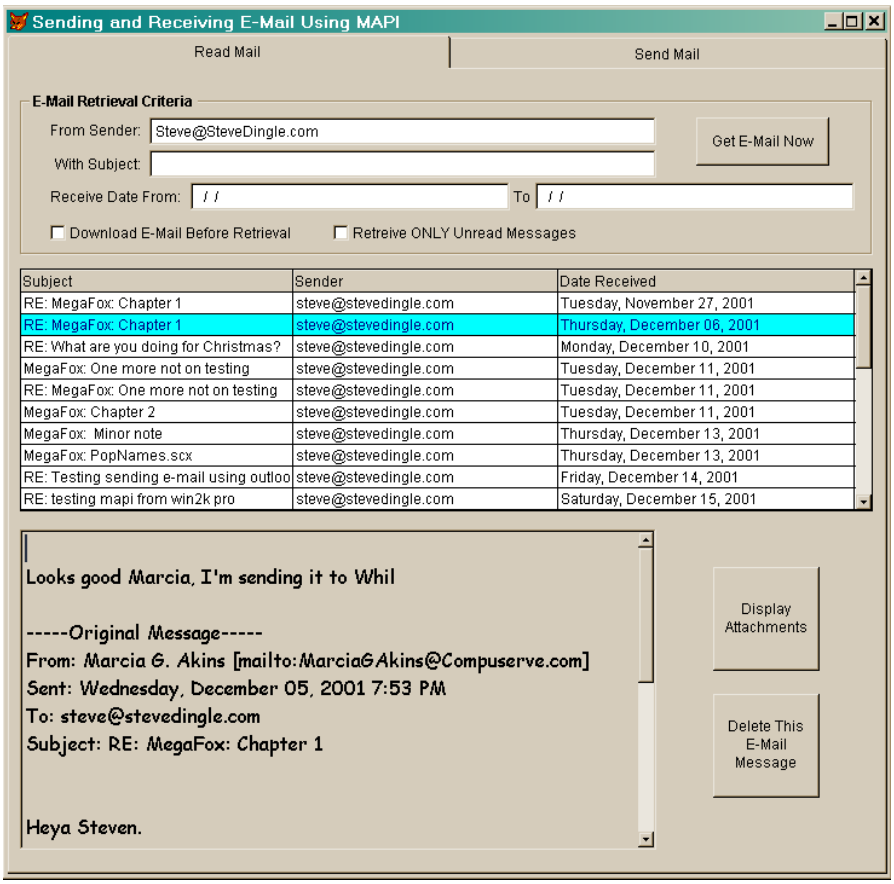


***Figure 2**. Example form for receiving e-mail using MAPI.*

The form's *ReadMail()* method first clears the grid's cursor of any existing data and then calls the *ReadMail()* method of the *oMapi* object, passing a reference to *oReadParms*. The *oMapi* object returns the number of messages retrieved (or -1 if an error occurred), and the subsequent actions depend on what was returned. If any messages are retrieved, the subject, sender's name, and receipt details are inserted into the grid for display; otherwise, an appropriate message is displayed like this:

```
LOCAL lnMsgs, lnCnt
*** Empty the grid of any current messages
ZAP IN csrMapiMail
```

```
*** Call upon the MAPI class to read the specified messages
WITH This.oMapi
  lnMsgs = .ReadMail( This.oReadParms )
  *** Returns the number of messages retrieved if no errors
  *** -1 if an error condition
  DO CASE
  CASE lnMsgs > 0
    *** Populate the cursor for the grid's recordSource
    FOR lnCnt = 1 TO lnMsgs
      .GetMsg( lnCnt )
      INSERT INTO csrMapiMail ( cSubject, cSender, dReceived ) ;
          VALUES ( .GetSubject(), .GetSender(), .GetDateReceived() )
    ENDFOR
  CASE lnMsgs = 0
    MESSAGEBOX( 'There are no messages to read', 48, 'Major WAAAHHH!' )
  OTHERWISE
    MESSAGEBOX( 'Unable to read the mail at this time', 16, 'Major WAAAHHH!' )
  ENDCASE
ENDWITH

*** Now go to the first message
GO TOP IN csrMapiMail
WITH This.pgfMapiMail.pgReadMail
  WITH .grdCsrMapiMail
    .SetFocus()
    .RefreshControls()
  ENDWITH
  *** See if we can enable the 'Display Attachments'
  *** And 'Delete' buttons
  IF RECCOUNT( 'csrMapiMail' ) > 0
    .cmdDelete.Enabled = .T.
    .cmdDisplayAttachments.Enabled = .T.
  ELSE
    .cmdDelete.Enabled = .F.
    .cmdDisplayAttachments.Enabled = .F.
  ENDIF
ENDWITH
```

This code, in the grid's custom *RefreshControls()* method, ensures that the message pointed to by the current grid row is the current one in the *MapiMessages* object:

```
*** Refresh the contents of the edit box
*** With the body text of the current message
WITH Thisform.oMapi
  IF RECCOUNT( This.RecordSource ) > 0
    *** Make sure we are on the correct message in the message store
    .GetMsg( RECNO( This.RecordSource ) )
    *** Get the body text
    This.Parent.edtBodyText.Value = .GetBodyText()
  ELSE
    This.Parent.edtBodyText.Value = ''
  ENDIF
ENDWITH
```

How do I send mail using MAPI? (*Example: MapiMail.scx and CH04.vcx::cntMapi*)

Sending e-mail using MAPI is quite straightforward. All you need to do is call the *Compose()* method of the MAPI Messages object. This creates a new message and points the Messages object to it by setting its *MsgIndex* property to -1. Once a new message is in the buffer, you need to populate its *Subject* property and add at least one recipient. This is all that is required before sending the sending the message.

How do I add recipients to a message?

The MAPI Messages object has five properties that are used to get and set the recipients of the current message:

- *RecipCount* — The number of recipients. This property is automatically updated as you add recipients, so there is never any need to increment it explicitly.
- *RecipIndex* — A zero-based pointer to the current recipient. This property behaves much like the *MsgIndex* property. Setting the *RecipIndex* of the MAPI Messages object causes all of the recipient-related properties to be updated with the details of the current recipient. To add a new recipient, all you have to do is set the *RecipIndex* to a value equal to the message's *RecipCount* (remember, this index is zero-based!).
- *RecipDisplayName* — The display name of the current recipient. This can be either a display name from the client's address book or an e-mail address.
- *RecipAddress* — The e-mail address of the current recipient. Filled in by MAPI when it resolves the *RecipDisplayName*.
- *RecipType* — The type of recipient. Allowable values are 1-To, 2-CC, and 3-Bcc.

So, for example, if there is already a message in the compose buffer, use code like this to add Andy Kramek as a recipient on the "To" list:

```
WITH Thisform.oMapi.oMessage
  .RecipIndex = .RecipCount
  .RecipDisplayName = "AndyKr@Compuserve.com"
  .RecipType = 1
ENDWITH
```

Although *RecipDisplayName* will accept either the display name from the address book or an e-mail address, we strongly advise sticking to e-mail addresses when automating e-mail in case a person has multiple e-mail addresses. MAPI simply throws an error when confronted with that dilemma!

How do I add attachments to a message?

You should be starting to see a pattern here because the way that attachments are added to an outgoing message is very similar to the way recipients are added. Just set the *AttachmentIndex* property of the MAPI Messages object to a value that is equal to its *AttachmentCount*. Then set the *AttachmentPathName* property. If you want the display name of the attached file to be different from the actual file name, you can set the *AttachmentName* property, but this is not required. The only thing that is a little tricky is setting the *AttachmentPosition* property of the current attachment. If you do not calculate this properly, the attachments wind up replacing text in the middle of your message!

You may be thinking that the obvious solution is to add attachments at the end of the message. The snag is that MAPI will not let you do it. The *AttachmentPosition* must specify a character position *within* the message, and the attachment is inserted, replacing whatever is at the specified position. The easiest way to resolve this is to add a couple of carriage returns and enough blank spaces to accommodate the attachments to the end of the message body. So, if you want to add three attachments to the end of a message, first add the required space to the message body like this:

```
WITH Thisform.oMapi.oMessage
  .MsgNoteText = .MsgNoteText + CHR( 13 ) + CHR( 13 ) + SPACE( 5 )
ENDWITH
```

Now, when you start to add attachments, all you have to do is to assign the first one an *AttachmentPosition* equal to the length of the original message incremented by three (for the two carriage returns and a space before the first attachment). That is how we handle it in our *cntMapi* class.

How does the custom *cntMAPI* class simplify sending e-mail?

Sending e-mail using MAPI is more straightforward than reading it, and our *cntMapi* class makes it even easier. Its custom *SendMail()* method does all the work required to create and send the message. Like the *ReadMail()* method discussed earlier, *SendMail()* expects a single parameter object. We use the *MapiSendParms* class to pass the necessary values, which are then used by *SendMail()* to create a new message by populating properties of the *MapiMessages* object. The properties are described in **Table 3**.

***Table 3**. Properties of the MAPI send mail parameter object.*

Property	Description
aAttachments	Array that holds the fully qualified name of all files to attach to the message.
aRecipients	Array that holds the e-mail addresses of all recipients of the message and the recipient type. Recipient types are: 1 = Main, 2 = CC, 3 = BCC.
cBodyText	The actual message text.
cPassword	Password associated with current MAPI session.
cSubject	Subject line for the e-mail.
cUserName	User name associated with the current MAPI session.
lDownload	When true, downloads new mail before processing.

Just as on the "Read Mail" page, the subject and message controls on the "Send Mail" page of the sample form (see **Figure 3**) are bound directly to the properties of the parameter object—in this case, *oSendParms*.

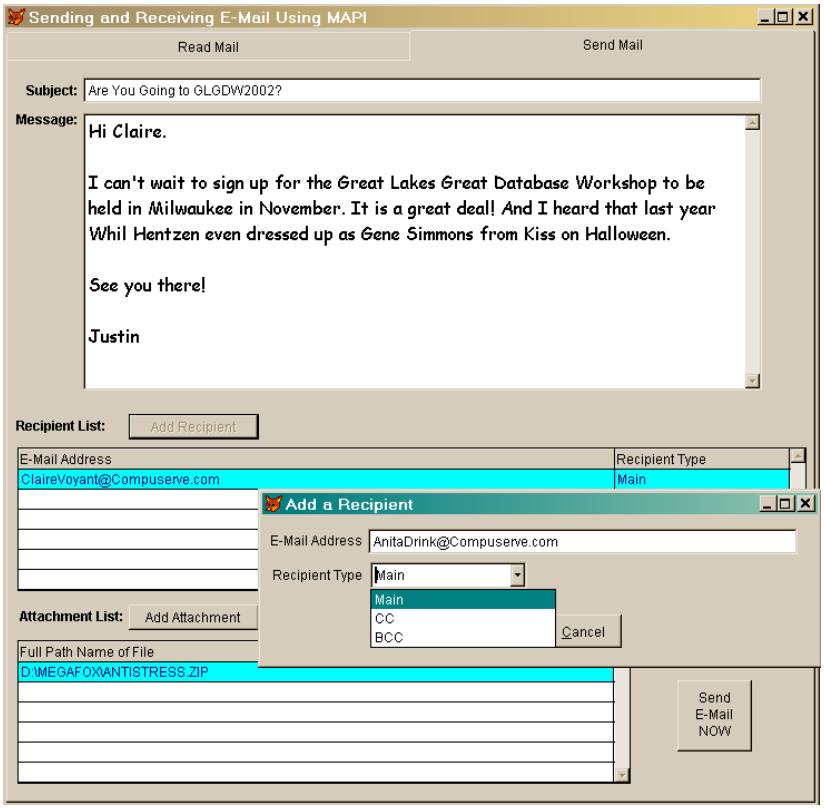


***Figure 3**. Demonstration form for sending e-mail using MAPI.*

When the "Send E-Mail NOW" button is clicked, the form's custom *SendMail()* method populates the *aRecipients* and *aAttachments* properties of the parameter object and then calls the *SendMail()* method of the MAPI container class.

```
*** Populate the array properties of the send parameters object
*** with the contents of the recipients and attachments cursors
SELECT * FROM csrRecipients INTO ARRAY Thisform.oSendParms.aRecipients
SELECT * FROM csrAttachments INTO ARRAY Thisform.oSendParms.aAttachments
Thisform.oMapi.SendMail( Thisform.oSendParms )
```

This code, in the *SendMail()* method of the MAPI container class, populates the *MAPIMessages* object and sends the message:

```
*** Create message and send it
WITH THIS.oMessage
  .SessionID = lnSessionID
```

```
  .Compose()

  *** Make sure we have enough room to add the attachments
  *** on to the end of the body
  .MsgNoteText = toSendParms.cBodyText + CHR(13) + CHR(13) + ;
                 SPACE( ALEN( toSendParms.aAttachments, 1 ) + 2 )
  .MsgSubject = toSendParms.cSubject

  *** Add the recipients
  *** The e-mail address is column 1
  *** The recipient type is column 2
  FOR lnCnt = 1 TO ALEN( toSendParms.aRecipients, 1 )
    .RecipIndex = .RecipCount
    .RecipDisplayName = ALLTRIM( toSendParms.aRecipients[ lnCnt, 1 ] )
    .RecipType = toSendParms.aRecipients[ lnCnt, 2 ]
  ENDFOR

  *** Finally add the attachments
  *** find the correct position for the first one
  lnPos = LEN( toSendParms.cBodyText ) + 3
  IF NOT EMPTY( toSendParms.aAttachments[ 1 ] )
    FOR lnCnt = 1 TO ALEN( toSendParms.aAttachments, 1 )
      .AttachmentIndex = .AttachmentCount
      .AttachmentPosition = lnPos
      .AttachmentName = JUSTFNAME(ALLTRIM(toSendParms.aAttachments[ lnCnt ]))
      .AttachmentPathName = ALLTRIM( toSendParms.aAttachments[ lnCnt ] )
      lnPos = lnPos + 1
    ENDFOR
  ENDIF

  *** All systems go: send the e-mail
  *** An argument of 1 will open client to manually send composed message
  .Send( 0 )

  *** Sign off
  This.oSession.SignOff()
ENDWITH
```

One problem that we noticed was that when the e-mail client was Outlook 2000 or later, invoking the *Send()* method of the MAPI Messages object did not actually send the e-mail. All it did was put the message into the outbox. In order to send the message, we had to do it manually. When Outlook Express was the default client, MAPI respected its configuration. If Outlook Express was configured to send mail immediately, the message was sent immediately; otherwise, it was placed in the outbox. We could not make Outlook 2000 behave as nicely, no matter how it was configured.

What is CDO 2.0?

Unlike earlier versions, CDO 2.0 (or CDO for Windows 2000) is not MAPI-based. It sends messages using the SMTP and/or NNTP protocols across the network, or through the pickup directory of a local SMTP or NNTP service. CDO 2.0 provides functionality that is simply not available using MAPI. For example, the message body is no longer limited to sending simple text messages and can include formatted HTML and even entire Web pages.

It consists of a single COM component (CDOSYS.DLL on Windows 2000 and CDOEX.DLL on Windows XP) that provides the tools to send messages formatted as either simple text or

according to the Multipurpose Internet Mail Extensions (MIME) specification. You can also intercept messages that arrive at a local SMTP or NNTP service and take specific action based on message content. This ability has some very important implications if you are hosting your own mail server. For example, it allows you to:

- Reject spam
- Check inbound messages for viruses
- Redirect a message from its original delivery path

While this is all very interesting, it does require that you have your own local mail server and is, therefore, way beyond the scope of this book. If you are doing your own hosting (or have your own mail server) and need to know more, the Platform SDK for CDO for Windows 2000 (CDOSYS.CHM) that ships with the MSDN library is an excellent source of information.

How do I send mail using CDO 2.0? (*Example: CdoMail.scx and CH04.vcx::cusCdo*)

To send messages, you must have network or local access to an SMTP or NNTP service. You must also have CDOSYS.DLL and Microsoft ActiveX® Data Objects (ADO) 2.5, or later, installed. CDO for Windows 2000 is fully integrated with ADO, providing a consistent interface for managing the data that comprises a message. (Note that ADO is installed by default with Windows 2000, but CDO is an optional item.) If CDO is installed on your machine, you will find either CDOSYS.DLL (Windows 2000) or CDOEX.DLL (Windows XP) in the C:\WinNT\System32 directory.

You must instantiate two objects to send mail using CDO: *CDO.Configuration* and *CDO.Message*. The *CDO.Configuration* object defines how messages are transmitted. If you do not have the Simple Mail Transport Protocol (SMTP) service installed locally, the message must be configured to use an SMTP service on the network. The *CDO.Configuration* object is loaded with the default configuration information. The exact values depend on the software that is installed on the local machine, but it includes, among others, the following items:

- Name of the SMTP server on the network
- SMTP server port
- SMTP account name
- Sender e-mail address
- Sender user name for the SMTP server
- Sender password for the SMTP server

The easy way to make sure that the *CDO.Configuration* object gets loaded with the required information is to install Outlook Express and configure it as an e-mail client, even if you never use it. Otherwise, you need to store the information somewhere and configure this object manually.

The *CDO.Message* object defines the actual message that is to be sent. In order to send a message, the "*Configuration*" property must be set to point to the Configuration object. In addition, the following properties on the Message object must be populated. At least one addressee must be specified, along with the subject line and the message body. All other properties are optional and depend on the content of the message. The most important properties of the Message object are:

- *To* — A comma separated list of e-mail addresses for the main recipients of the message.
- *Cc* — A comma separated list of e-mail addresses for the carbon copy recipients.
- *Bcc* — A comma separated list of e-mail addresses for the blind carbon copy recipients.
- *HtmlBody* — The HTML formatted representation of the message.
- *TextBody* — The plain text representation of the message. When the *AutoGenerateTextBody* and *MimeFormatted* properties of the Message object are both true and you set *HTMLBody*, CDO automatically sets the *TextBody* property to the plain text equivalent.

Notice that there are actually two properties that refer to the message body. Using CDO it is possible to send multi-part messages that contain both plain text and HTML. The interaction between these properties is complex, and is governed by the *AutoGenerateTextBody* property of the Message object. If you need this degree of complexity, you will need to refer to the CDOSYS.CHM help file for details.

There are two important *CDO.Message* methods, in addition to *Send()*:

- *CreateMHTMLBody* — Converts the contents of an entire Web page into a MIME Encapsulation of Aggregate HTML Documents formatted in the message body. In doing so it replaces any previous contents of the *HTMLBody*.
- *AddAttachment* — Adds attachments to the message. Requires the fully qualified path name (or URL) of the file to be attached. If you populate the *HTMLBody* before calling *AddAttachment()* with a URL, any in-line images are displayed as part of the message.

How does the *cusCDO* class work?

For consistency with the approach we took to MAPI, we have implemented two classes for sending mail with CDO: *cusCdo*, which does the work, and *cdoParms*, which is the parameter object. You may wonder why we used a custom class for our CDO wrapper instead of the container that we used for MAPI. The answer is simple. We used a container for our custom

MAPI class because the MAPISession and MAPIMessages objects are ActiveX controls that can be dropped into a container visually in the Class Designer. The CDO Configuration and Message objects are classes inside a DLL and must be instantiated using **CREATEOBJECT()** or **ADDOBJECT()**. Using a container class to wrap CDO provided no benefit.

The example form has an instance of the *cusCdo* class (called "*oCDO*"), and the form controls are bound directly to the properties of the parameter object (called "*oParms*"). The properties are listed in **Table 4**.

***Table 4**. Properties of the CDO parameter object.*

Property	Description
aAttachments	Array that holds the fully qualified name of all files to attach to the message.
cBcc	Identifies blind carbon copy recipients for the message.
cCC	Identifies secondary or carbon copy recipients for the message.
cFrom	Identifies the sender of the message.
cHTMLBody	The Hypertext Markup Language (HTML) representation of the message.
cFrom	Identifies the person or entity that actually submitted the message if this person or entity is not the sole identity in the From header.
cSubject	Identifies the subject of the message.
cTo	Identifies the primary recipients of the message.
curl	Used by the *CDO.message* object's *CreateMHTMLBody()* method to convert the contents of an entire Web page into a MIME Encapsulation of Aggregate HTML Documents (MHTML) formatted message body.

The form has a custom *SendMail()* method that is called when the "Send E-Mail NOW" button is clicked.

This method first verifies, at least superficially, that if a Web page was specified for inclusion in the message body, it is indeed a valid URL. It then populates the parameter object's *aAttachments* property with all attachments to be included in the message and passes it to the CDO object's *SendMail()* method like this:

```
IF NOT EMPTY( Thisform.oParms.cUrl ) AND ;
   UPPER( LEFT( ALLTRIM( Thisform.oparms.cUrl ), 7 ) ) # "HTTP://"
  MESSAGEBOX( 'You can only include the contents of valid web pages',;
              16, 'Please Fix Your Input and try again' )
  Thisform.txtcURL.SetFocus()
ELSE
  *** Get the attachment files (if any) into the parameter object
  SELECT * FROM csrAttachments INTO ARRAY Thisform.oParms.aAttachments
  Thisform.oCDO.SendMail( Thisform.oParms )
ENDIF
```

The custom *SendMail()* method of our CDO wrapper sets properties of the *CDO.Message* object with the information in the parameter object. Once this is done, all that is left to do is to call the *Send()* method of the Message object.

```
WITH This.oMsg
  .Configuration = This.oConfig
  *** See if we have a sender address.
  *** If we manually loaded the config, we may not
  IF EMPTY( NVL( .From, "" ) )
```

```
    .From = .Configuration.Fields( cdoSendEmailAddress ).value
  ENDIF
  .To = ALLTRIM( toParms.cTo )
  .CC = ALLTRIM( toParms.cCC )
  .Bcc = ALLTRIM( toParms.cBcc )
  .Subject = ALLTRIM( toParms.cSubject )

  *** See if we are sending a web page in the body of the message
  IF NOT EMPTY( toParms.cURL )
    .CreateMHTMLBody( ALLTRIM( toParms.cURL ) )
  ENDIF

  *** Add any message text to the beginning of the body
  .HTMLBody = toParms.cHTMLBody + .HTMLBody

  *** Add any attachments
  IF NOT EMPTY( toParms.aAttachments[ 1 ] )
    lnLen = ALEN( toParms.aAttachments, 1 )
    FOR lnCnt = 1 TO lnLen
      .AddAttachment( ALLTRIM( toParms.aAttachments[ lnCnt ] ) )
    ENDFOR
  ENDIF
  .Send()
ENDWITH
```

When the *cusCDO* class is instantiated, the object instantiates the *CDO.Configuration* and *CDO.Message* objects that are required to send mail.

```
WITH This
  *** create configuration and message objects
  .oConfig = CREATEOBJECT( 'CDO.Configuration' )
  IF TYPE( 'This.oConfig' ) = 'O'
    *** Check to see if we have configuration infomation
    IF NOT EMPTY( NVL( .oConfig.Fields( ;
     "http://schemas.microsoft.com/cdo/configuration/smtpserver").value, "" ) )
      llRetVal = .T.
    ELSE
      *** Manually set Configuration properties
      *** using the CdoConfig table
      llRetVal = This.GetSmtpInfo()
    ENDIF
  ENDIF
  IF llretVal
    .oMsg = CREATEOBJECT( 'CDO.Message' )
    llretVal = IIF( TYPE( 'This.oMsg' ) = 'O', .T., .F. )
  ENDIF
ENDWITH
RETURN llRetVal
```

The sample code uses a table called CDOCONFIG.DBF to store the configuration information and the custom *GetSmtpInfo()* method uses it to manually configure CDO.

```
SELECT CdoConfig
SCAN
  IF NOT EMPTY( CdoConfig.cVal )
    lcFieldName = ALLTRIM( cdoConfig.cFld )
    This.oConfig.Fields( lcFieldName ).Value = ;
```

```
        This.Str2Exp( ALLTRIM( CdoConfig.cVal ), cType )
  ENDIF
ENDSCAN
This.oConfig.Fields.Update()
```

The e-mail sent from the form in **Figure 4** then looks like **Figure 5**.

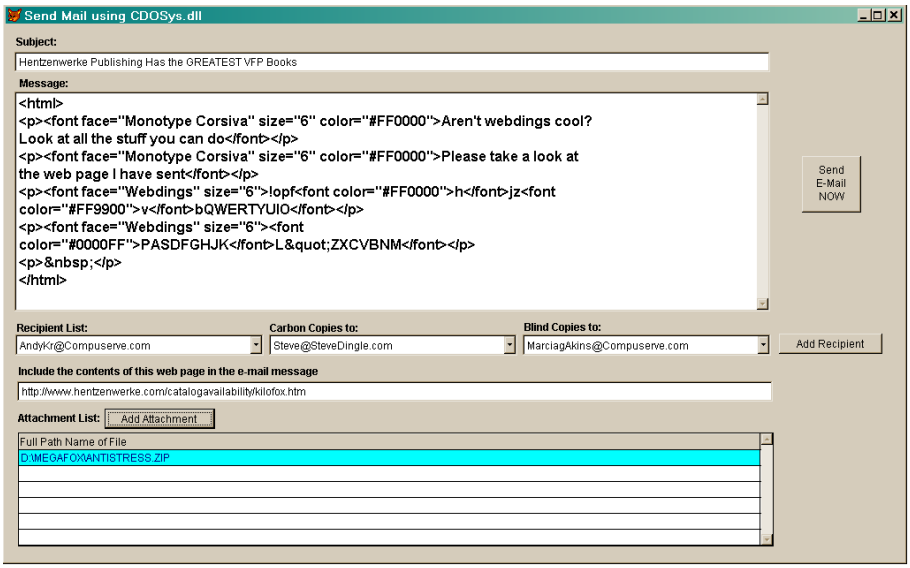


***Figure 4**. Demonstration form for sending e-mail using CDO for Windows 2000.*

***Figure 5**. You can send very complex messages using CDO for Windows 2000.*

Can I control Outlook programmatically? (*Example: OutlookMail.scx and CH04.vcx::cusOutlook*)

The Outlook object model provides a rich and powerful interface for, among other things, sending and receiving e-mail. As an Automation server, it exposes its interface through COM, which means that anything that you can do interactively you can also do programmatically. You can even access the Outlook address book, something that you cannot do when using simple MAPI or CDO. A complete discussion on all the possibilities that Outlook Automation offers is clearly beyond the scope of a chapter that is limited to e-mail. However, there is good documentation of Outlook's exposed objects, properties, events, and methods in the *Office 2000 Language Reference*. The book *Microsoft Office Automation with Visual FoxPro* by Tamar Granor and Della Martin (Henztenwerke Publishing, 2000, ISBN: 0-9655093-0-3) also covers the Outlook object model.

To automate Outlook, the first thing you need to do is create an instance of the Outlook *Application* object. Next, you need to get access to its data. However, you cannot get direct access and must create the *Namespace* object that acts as a gateway. The *Namespace* provides methods for logging into, and out of, a data source as well as some additional data source specific methods. Currently the only data source supported by Outlook is the "MAPI" data source, and its *Namespace* object provides, among other things, methods for accessing Outlook's special folders directly. This is handled in the custom *CreateSession()* method of our wrapper class as follows:

```
*** See if we already have an instance of Outlook Running
IF TYPE( 'This.oOutlook' ) = 'O' AND NOT ISNULL( This.oOutlook )
  *** No need to create a new instance
ELSE
  WITH This
    .oOutLook = CREATEOBJECT( 'Outlook.Application' )
    IF TYPE( 'This.oOutLook' ) = 'O' AND NOT ISNULL( .oOutlook )
      .oNameSpace = .oOutlook.GetNameSpace( 'MAPI' )
      IF TYPE( 'This.oNameSpace' ) = 'O' AND NOT ISNULL( .oNameSpace )
        llRetVal = .T.
      ENDIF
    ENDIF
  ENDWITH
ENDIF
```

The first problem that we encountered when automating Outlook was that "magic number" constants are used extensively to define the elements of its various collections (folders, recipients, attachments, and so on) and to attribute meanings to properties. For example, if the *class* of a *Contact* object is 40, it is a "contact" not a "distribution list". This methodology also means that to get a reference to the inbox we need code like this:

```
loInbox = This.oNameSpace.GetDefaultFolder( 6 )
```

The constants for the rest of Outlook's default folders are listed in **Table 5**.

***Table 5**. Outlook constants used by GetDefaultFolder.*

Folder name	Outlook constant in include file	Constant value
Deleted Items	olFolderDeletedItems	3
Outbox	olFolderOutbox	4
SentMail	olFolderSentMail	5
Inbox	olFolderInbox	6
Calendar	olFolderCalendar	9
Contacts	olFolderContacts	10
Journal	olFolderJournal	11
Notes	olFolderNotes	12
Tasks	olFolderTasks	13
Drafts	olFolderDrafts	16

To make our lives easier, even though we do not normally like them, we created an include file for these constants. This file, *msoutl9.h*, was created by using Rick Strahl's *GetConstants* program (which can be downloaded free at www.west-wind.com/Webtools.asp).

The second problem encountered was that all Outlook collections (for instance, *Folders*, *Attachments*, *Recipients*) have an *Items* collection. This is helpful because it provides a consistent interface for iterating the items contained in any folder. However, these different *Items* collections don't share the same methods and properties. For example, the *Items* in the *Inbox* (e-mail messages) do not expose the same properties as the *Items* in the *Contacts* folder (contacts). Trying to discover which properties were exposed by a specific *Items* collection was a painful process and the object browser just wasn't much help. We found it was simpler to instantiate the objects in which we were interested in the command window and to use IntelliSense to reveal their mysteries.

How do I access the address book?

The address book is just another object. More specifically, it is the *Contacts* folder. So all you need to do is get a reference to it and iterate through its *Items* collection, storing the information that you are interested in. This is exactly what the custom *GetContacts()* method of our Outlook wrapper class does. It stores the names and e-mail addresses of all the contacts in an internal array so that they are available for the entire lifetime of the object.

```
LOCAL loAddressBook AS Outlook.MAPIFolder, loContact AS Object, lnContactCount

*** Get a reference to the contacts folder
loAddressBook = This.oNameSpace.GetDefaultFolder( olFolderContacts )
IF VARTYPE( loAddressBook ) = 'O'
  lnContactCount = 0

  *** Get info about each contact into the array
  FOR EACH loContact IN loAddressBook.Items
    WITH loContact

      *** Make sure we only get individual contacts
      *** and skip any distribution lists
      IF .Class = olContact
        lnContactCount = lnContactCount + 1
        DIMENSION This.aContacts[ lnContactCount, 4 ]
        This.aContacts[ lnContactCount, 1 ] = .LastName
```

```
        This.aContacts[ lnContactCount, 2 ] = .FirstName
        This.aContacts[ lnContactCount, 3 ] = .Email1Address
        This.aContacts[ lnContactCount, 4 ] = .FullName
      ENDIF
    ENDWITH
  ENDFOR
  ASORT( This.aContacts )
ENDIF
```

How do I read mail using Outlook Automation?

First you need to use the *NameSpace* object's *GetDefaultFolder()* method, with the correct constant (see Table 5) to obtain an object reference to the Inbox. Then it is a simple matter to iterate through its *Items* collection, accessing the relevant properties of each item.

It is even possible to retrieve only messages that satisfy some filter condition, such as from a particular sender or received within a specified date range. This can be done in two ways, first by using the *Restrict()* method. This returns a new collection containing only those items that match the filter. For example, to retrieve only unread messages, use this syntax:

```
loMessages = loInbox.Items.Restrict( "[Unread] = True" )
```

The alternative to using *Restrict()* is to use *Find()* in conjunction with *FindNext()* to iterate through the *Items* collection. This method offers better performance than the *Restrict()* method when dealing with small collections. To iterate through all the unread messages using the *Find()* method, use this syntax:

```
loMsg = loInbox.Items.Find( "[Unread] = True" )
DO WHILE VARTYPE( loMsg ) = 'O'
  *** Call a method that processes the current message
  This.ProcessMessage( loMsg )
  loMsg = loInbox.Items.FindNext()
ENDDO
```

Keep in mind that you cannot perform searches of the type that you can in Visual FoxPro with `SET( "EXACT" ) = OFF` when using either *Restrict()* or *Find()*. For example, you cannot find all the messages with a subject line that starts with "RE: MegaFox" by using either of these two methods. The only solution is to write code that iterates through the messages and compares the subject line of each to the required string.

When reading e-mail we are, obviously, interested only in the *Inbox* folder, whose *Items* collection has so many properties that we cannot possibly list all of them here. The following are the most important:

- *Attachments* Collection of attachment files belonging to the current *Item*.
- *To* Comma separated list containing the display names of the recipients in the To List.
- *Cc* Comma separated list containing the display names of the carbon copy recipients.

- *Bcc* Comma separated list containing the display names of the blind carbon copy recipients.
- *Recipients* Collection of all recipients of the current item. Each *Recipient* item has a type property that specifies whether it is a *To, Cc*, or *Bcc*.
- *SenderName* Display Name of the person who sent the item.
- *Subject* Message subject line.
- *ReceivedTime* Date and time that the message was received. Note that, even though this property is of data type DateTime, when using this property to *Restrict()* messages retrieved, the filter condition must be specified in string form (for example, "[ReceivedTime] < '12/25/2001'")
- *Body* The plain text body of the message.
- *HTMLBody* The HTML formatted body of the message.

Remember that although all Outlook folders have an *Items* collection, not all *Items* collections have the same set of properties. Just because the *Items* collection of the Inbox has an *Attachments* collection, there is no guarantee that the *Items* collection of any other folder will have one.

How does the *ReadMail()* method work?

For consistency with the approach we took to MAPI, we have implemented two classes for reading mail with Outlook Automation: *cusOutlook*, which does the work, and *OutlookReadParms*, which is the parameter object. The example form has an instance of the *cusOutlook* class (called *oMail)* and the form controls are bound directly to the properties of the parameter object (called *oReadParms*). The properties of the parameter object are used to obtain a filtered subset of the messages in the inbox and are almost identical to those of the *MapiReadParms* object.

As a matter of fact, both classes (*cntMapi* and *cusOutlook*) share a very similar public interface, so the two sample forms are virtually identical. The only difference between the two forms is the button on OUTLOOKMAIL.SCX that allows the user to display the contact list. Simple MAPI provides only the crudest mechanism for accessing the e-mail client's address book: You can call the *Show()* method of the *MapiMessages* object to view the e-mail client's address book. However, you can only view it and cannot retrieve any data from it, so it is of limited usefulness.

The *ReadMail()* method expects a single parameter object that is populated with any filter conditions to apply to the messages to be retrieved from the Outlook *Inbox*. This parameter object can be configured to retrieve:

- Unread messages
- Messages from a specific sender

- Messages received within a certain date range
- Messages with a specific subject line

Any combination of these filters can be applied to determine which messages are retrieved by the method.

After the parameter object is validated, the parameter object is passed to the custom *BuildFilter()* method. This method concatenates the first three filter conditions and returns them as a single string. It then obtains an object reference to the Outlook *Inbox* and uses it to retrieve a set of messages. This is accomplished by passing the filter condition to the *Restrict()* method of *Inbox*'s *Items* collection. We then iterate through the messages returned and check the subject line of each one individually. If the message has the specified subject line, it is added to *cusOutlook*'s internal array of messages (*aMsgs)*. Finally, the number of messages retrieved (or -1 if an error occurred) is returned to the caller.

```
*** Build the filter condition
lcFilter = This.BuildFilter( toReadParms )

*** Get an object reference to the inbox
*** The constant 'olFolderInbox' is in the msoutl9.h include file
*** along with all the other outlook constants
loInbox = This.oNameSpace.GetDefaultFolder( olFolderInbox )
IF NOT EMPTY( lcFilter )
  loMessages = loInbox.Items.Restrict( lcFilter )
ELSE
  loMessages = loInbox.Items
ENDIF

*** Go through the collection of messages retrieved
*** and save only the ones we are interested in to the aMsgs array
*** the Restrict method doesn't work consistently with restricting
*** messages by subject
IF VARTYPE( loMessages ) = 'O'
  lnMsgCount = 0
  lcSubject = UPPER( ALLTRIM( toReadParms.cSubject ) )
  FOR lnMsg = 1 TO loMessages.Count
    IF NOT EMPTY( lcSubject )
      IF UPPER( ALLTRIM( loMessages.Item[ lnMsg ].Subject ) ) = lcSubject
        lnMsgCount = lnMsgCount + 1
        DIMENSION This.aMsgs[ lnMsgCount ]
        This.aMsgs[ lnMsgCount ] = loMessages.Item[ lnMsg ]
      ENDIF
    ELSE
      lnMsgCount = lnMsgCount + 1
      DIMENSION This.aMsgs[ lnMsgCount ]
      This.aMsgs[ lnMsgCount ] = loMessages.Item[ lnMsg ]
    ENDIF
  ENDFOR
ELSE
  lnMsgCount = -1
ENDIF
RETURN lnMsgCount
```

How do I send mail using Outlook Automation?

Sending mail using Outlook Automation is a very simple process indeed. Once we instantiate the *Outlook.Application* object and obtain a reference to the MAPI *Namespace* (just as we do when reading e-mail), all that is required is to invoke Outlook's *CreateItem()* method to create a new message object. Once you have populated the necessary properties, a call to the object's *Send()* method is all that is required.

Our Outlook Automation class, like the other classes in this chapter, has a single custom *SendMail()* method that does all the work required to create and send the message. As usual, *SendMail()* expects a single parameter object. We use the *OutlookSendParms* class to pass the necessary values, which are then used by *cusOutlook.SendMail()* to populate the properties of the newly created message.

```
*** Create a new mail item
loMsg = This.oOutlook.CreateItem( olMailItem )
IF VARTYPE( loMsg ) = 'O'
  WITH loMsg
    *** Set the required message properties
    .Subject = ALLTRIM( toSendParms.cSubject )
    .Body = ALLTRIM( toSendParms.cBodyText )

    *** Add the recipients
    lnLen = ALEN( toSendParms.aRecipients, 1 )
    FOR lnCnt = 1 TO lnLen
      .Recipients.Add( ALLTRIM( toSendParms.aRecipients[ lnCnt, 1 ] ) )
      .Recipients[ lnCnt ].Type = toSendParms.aRecipients[ lnCnt, 2 ]
    ENDFOR

    *** And finally, add the attachment if there are any
    IF NOT EMPTY( toSendParms.aAttachments[ 1 ] )
      lnLen = ALEN( toSendParms.aRecipients, 1 )
      FOR lnCnt = 1 TO lnLen
        .Attachments.Add( ALLTRIM( toSendParms.aAttachments[ lnCnt, 1 ] ) )
      ENDFOR
    ENDIF

    *** And send it off
    .Send()
  ENDWITH
ENDIF
```

Conclusion

This chapter has provided the details of several different mechanisms for sending and receiving e-mail. Which is the best choice? As usual, the answer is "it depends." In our opinion, CDO for Windows 2000 or later is the best solution for sending e-mail for a couple of reasons:

- It allows you to send formatted HTML and Web pages.
- It does not require REDEMPTION.DLL to work around the security patch to send e-mail without any user intervention.

But what if you need to send or receive e-mail from earlier Windows versions? At the time of this writing, you have a couple of options. You can use *wwipstuff* or you can download REDEPTION.DLL if you want to bypass the security patch and use MAPI or Outlook Automation. Whatever your decision, we hope that we have provided some elegant solutions for e-mail enabling your Visual FoxPro applications.

Chapter 5
Accessing the Internet

Visual FoxPro is well known as a superb desktop database and has traditionally been used for developing LAN-based, line-of-business, applications. However, it is capable of much more than that. This chapter shows how you can use the Microsoft Web Browser control, inside VFP, to gain access to information that is available on the World Wide Web. It also covers creating and implementing hyperlinks and accessing Web Services as further ways to extend the scope of your Visual FoxPro applications. (Note that creating Web Services in VFP is discussed in Chapter 16, which also includes building COM components.)

How do I show a Web page in a form? (*Example: frmBrow.scx*)

There is a very simple answer to this question—use the Microsoft Web Browser ActiveX control. This is one of the standard Windows ActiveX controls and is available on any machine with Microsoft Internet Explorer Version 3.0 or later.

To add an instance of the Web Browser ActiveX control, just drag an OLE Container control to the form. This brings up the ActiveX Selection dialog (see **Figure 1**). Find the Web Browser in the list, make sure that "Insert Control" is selected, and click OK. *Voilà!* You now have a browser inside a VFP form!

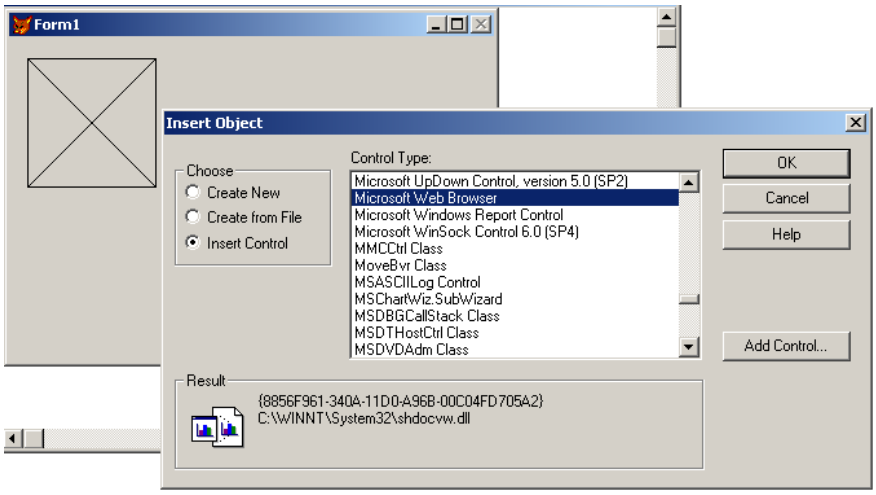


***Figure 1**. Adding the Web Browser control to an OLE Container control.*

But when I run the form, I get an error!

Ah yes. That is a small problem when you use this control! However, you will notice that it only happens once, and that the error occurs before the form is visible. It would seem,

although we cannot find any formal documentation to support this, that because the browser is an asynchronous control, it is attempting to communicate with the form before the form is actually available, hence the error. Fortunately, the solution is simple enough: Just add a **NODEFAULT** command to the *Refresh()* method of the control. This will not cause any problems because, in the context of a Visual FoxPro form, we want to control access to the *Refresh()* method explicitly anyway.

In fact, since we intend to use this control in several different scenarios, we created our own subclass of the control (see *CH05::xBrowser*) that has the *Refresh()* method set up correctly so that we no longer need to worry about it. This is the class that we have actually used in the sample form.

Displaying content

The Web Browser control is fully functional and is capable of displaying anything that can be handled by Internet Explorer. The usual context-sensitive shortcut menus are also available when you right-click on the control. All that is needed is to pass the location of the item that you want to display to the control's *Navigate()* method.

Note that the control has two methods concerned with navigation. The first, Navigate(), must receive as its first parameter a String expression that evaluates to either a URL, a fully qualified path and file name, or the Universal Naming Convention (UNC) location of the resource to display. The second method, Navigate2(), can handle all the same inputs but can also accept other formats, such as a pointer to an item identifier list (PIDL) for an entity in the Windows shell namespace. For more information on the Web Browser control, consult Microsoft Knowledgebase Article Q165212, which explains where to find the (very fragmented!) documentation. The Knowledgebase is available, online, at http://support.microsoft.com.

The example uses a simple free table, named SHOWDATA.DBF, to store the various locations that are then displayed in the grid on the first page of the form. Activating the second page loads the selected item into the browser control (see **Figure 2**). Of course, in order to actually display anything from the Web, you must have an Internet connection defined because the browser control uses the same settings as Internet Explorer. The code in the *Activate()* method of the second page merely performs some simple validation when an item of type **FILE** is passed:

```
LOCAL lcTarget, lcType
WITH This
  lcType = showdata.cType
  IF lcType = "URL"
    *** Just use the specified URL as is
    lcTarget = ALLTRIM( showdata.clocation )
  ELSE
    *** Browser needs a fully qualified Path/FileName
    lcTarget = FULLPATH( CURDIR()) + (ALLTRIM( showdata.clocation ))
  ENDIF
  *** Make sure that the specified file exists
  IF lcType = "FILE"
     *** Check that the file existS
```

```
      IF FILE( ALLTRIM( lcTarget ) )
        .oExplorer.Navigate( lcTarget )
      ELSE
        MESSAGEBOX( "Specified File: " + lcTarget ;
                          + "Does not exist", 16, "No can do!" )
        NODEFAULT
        This.Parent.ActivePage = 1
      ENDIF
  ELSE
    *** Just try and navigate to the specified location
    .oExplorer.Navigate( lcTarget )
  ENDIF
  *** Update Controls
  ThisForm.RefreshForm( .T. )
ENDWITH
```

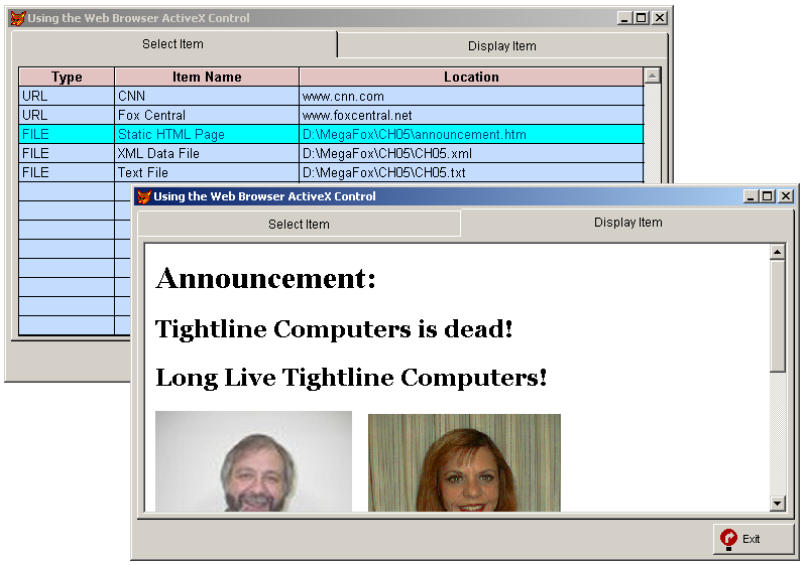


***Figure 2**. Using the Web Browser control to display an HTML file (frmbrow.scx).*

How do I put a browser on the VFP desktop? (*Example: frmDsktop.scx*)

The trick here is to use a form that looks just like the Visual FoxPro desktop and contains an instance of our Web Browser control, together with a couple of simple controls that allow us to enter a URL and navigate to it (see **Figure 3**). In order to make the form look like the normal desktop, we need to get rid of the title bar and borders so we have set the following properties:

- *BorderStyle = 0*—To remove all borders from the form.
- *TitleBar = OFF*—To remove the title bar, complete with the control box, maximize and minimize, help and close buttons.

- *WindowState = 2-Maximized*—To ensure that the form fills the available desktop.
- *AlwaysOnBottom = True*—To allow other windows to float over the browser so that we can continue working in VFP while the form is running.

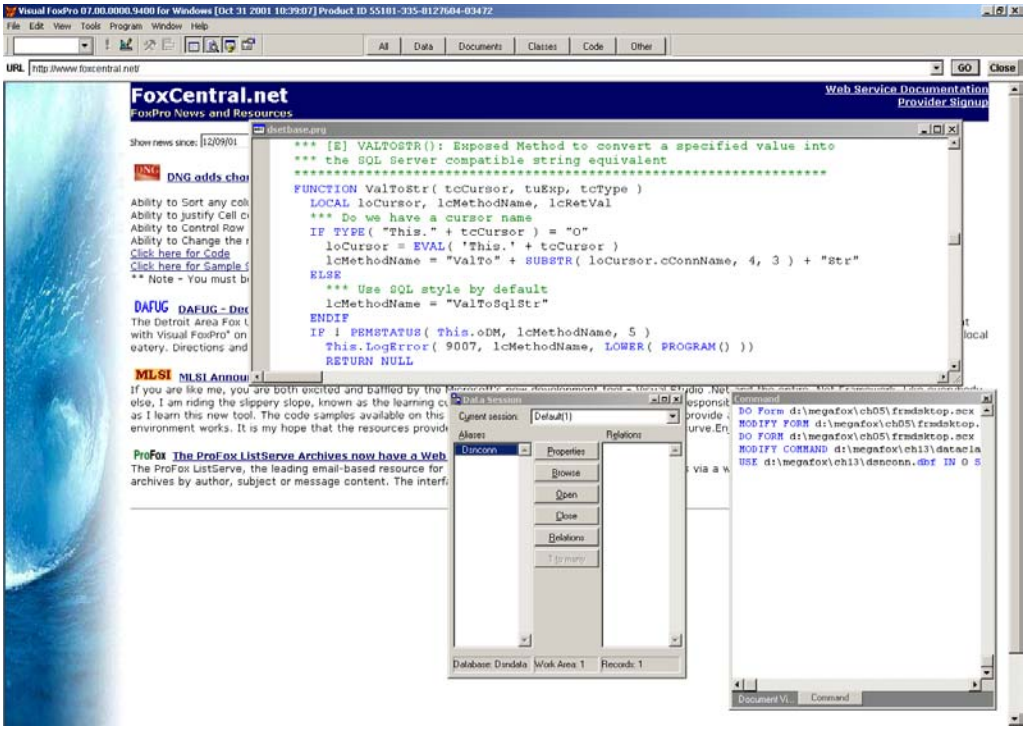


***Figure 3**. Using a Web Browser as your VFP Desktop (frmdsktop.scx).*

Functionally, this form is just an extension of the first use we made of the Web Browser control. Instead of pre-defining our locations in a table, the form has a combo box into which a location can be typed. (This has been set up so that as new items are typed they are added directly to the list, thereby providing a very simplistic "recall" facility.) A *Go* button calls the form's custom *Navigate()* method, which reads the value from the combo box and passes it on to the *Navigate()* method of the browser control.

One refinement is that the *Go* button's *Default* property has been set to True so that hitting Enter after typing something in the combo box also triggers the form's *Navigate()* method. The close button, as you would expect, calls the form's *Release()* method.

The only code needed (beside the *Init()* and *Resize()* methods, which merely size and re-position the controls using the current screen width as a guide) is in the form's *Navigate()* method. This reads the current value from the combo box and passes it on to the *Navigate()* method of the browser control:

```
LOCAL lcURL
*** Try to go there
WITH ThisForm
  *** Get URL from combo
  lcURL = ALLTRIM( .cboURL.DisplayValue )
  .oExplorer.Navigate( lcURL )
ENDWITH
```

As you can see, using the Web Browser control is really very simple and can provide a lot of functionality with very little code required.

How do I print the contents of a Web page? (*Example: frmBrow.scx*)

The Web Browser control makes this a simple task to execute, a little harder to figure out. The control exposes, in its interface, a method named *ExecWB()*. This method uses a COM interface (*IOLECommandTarget*) that allows us to execute any supported command remotely. The method's interface is defined as:

```
ExecWB( cmdID,  cmdopt[, *pvaIn[, *pvaOut]])
Where: cmdID (mandatory) is a defined command constant (OLECMDID)
       cmdOpt (mandatory) is a defined option constant (OLECMDEXECOPT)
       *pvaIn (Optional) pointer to structure with input arguments
       *pvaOut (Optional) pointer to structure for output results
```

That was the easy part. The hard part is that, in order to make use of this interface, we have to know what the values for the `OLECMDID` and `OLECMDEXECOPT` constants are! Unfortunately, these minor details are not actually documented anywhere that we could find, and there is no help file for the Web Browser control; that would be *too* easy! Fortunately we *can* get at them by examining the Type Library.

To do this you can either use the VFP Object Browser and drag the items that you want directly from the browser into a text file or, as we did, download Rick Strahl's free *GetConstants* utility from the West-Wind Web site (www.west-wind.com). This useful little tool prompts you for a type library and creates an include file that gives all the defined constants. (Note that you do need to know the actual file name on disk to use this utility. In this case the file is named SHDOCVW.DLL and it is installed in your \SYSTEM32\ subdirectory.) The sample code for this chapter includes the file OLECMDCONST.H that was produced using *GetConstants*.

Having gotten the list of available commands, we find that we can use *ExecWB()* to print the contents of the current document by simply calling it with `CMDID = 6` and `CMDOPT = 1` (display "Select Printer Dialog") or 2 (no dialog). The code in the form's custom *DoPrint()* method is therefore:

```
*** Call the Browser Control's Print method, prompt for printer
WITH ThisForm.pgfbrowser.Page2.oExplorer
  .execWB( OLECMDID_PRINT, OLECMDEXECOPT_PROMPTUSER )
ENDWITH
```

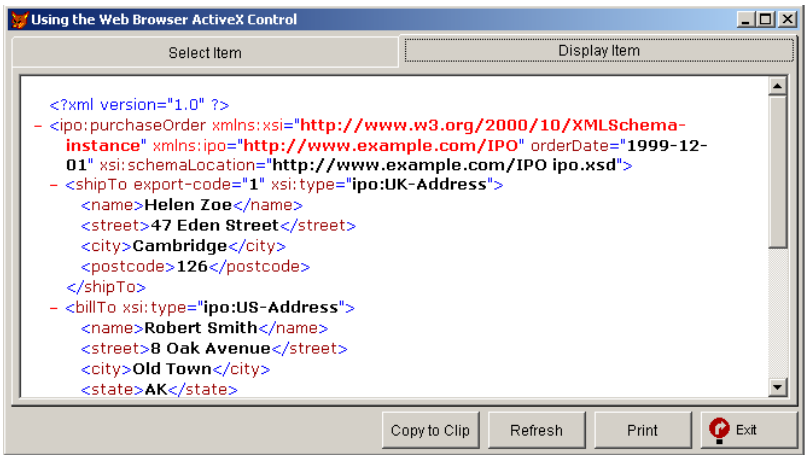


***Figure 4**. Added functionality with the browser control (frmbrow.scx).*

Similar methods, called by the control buttons on the example form, show how we use the *ExecWB()* method to Refresh the page and copy the contents of the current page to the clipboard (see **Figure 4**).

How do I extract data from a Web page? (*Example: frmKbase.scx*)

There are at least three ways of doing this, depending upon the degree of control that you need over what is extracted. First, we could use the Web Browser's own *ExecWB()* method and copy text out of the control. Second, we could get a reference to the document object, inside the Web Browser, and use its *ExecCommand()* method. Third, we could use the Document Object Model (DOM), which gives the greatest measure of control but is a little more complex. The example form illustrates this approach, but first we will cover the other options briefly.

Using the browser control's *ExecWB()* method

The browser control exposes, in its interface, a method named *ExecWB()* (see "How do I print the contents of a Web page?" for an explanation of how to call this method). An investigation of the pre-defined command constants in the OLECMDCONST.H file reveals that:

- 12 is defined as OLECMDID_COPY
- 17 is defined as OLECMDID_SELECTALL
- 18 is defined as OLECMDID_CLEARSELECTION

The mandatory Options parameter for all of these three commands can simply be passed as the "*do whatever is the default for this option*" value, which is 0.

So providing that the user has highlighted some text, we can use the Copy command to copy the highlighted area to the clipboard. Once there, the Visual FoxPro system variable *_ClipText* gives us access to it. This is simple and direct but does require that the user actually highlight something in the browser first. The alternative is to copy the entire contents of the

page, which we can do without user intervention by using the *SelectAll* and *ClearSelection* commands. This is exactly what this code, in the custom *DoCopy()* method called by the "Copy To Clip" button in the FRMBROW.SCX form, does.

```
*** Select all and copy to clipboard
*** Method [1] Using the ExecWB() method of the browser
WITH ThisForm.pgfbrowser.Page2.oExplorer
  .execWB( OLECMDID_SELECTALL, OLECMDEXECOPT_DODEFAULT )
  .execWB( OLECMDID_COPY, OLECMDEXECOPT_DODEFAULT )
  .execWB( OLECMDID_CLEARSELECTION, OLECMDEXECOPT_DODEFAULT )
ENDWITH
```

In fact, the *ExecWB()* method gives us access to a range of document-centric functions which, together with the other methods exposed in its interface, make managing the contents of the browser control programmatically a simple task. You can investigate the full set of properties, events, and methods available in the Web Browser control by examining the file SHDOCVW.DLL in the Object Browser.

Using the document object's *ExecCommand()* method

While the browser control's *ExecWB()* command does give us access to a lot of functionality, it is merely a wrapper around the document object's own *ExecCommand()* method. Full details of the Document Object Model can be found in the Help file (HTMLREF.CHM) that is installed, by default, in the subdirectory:

\Program Files\Microsoft Visual Studio\Common\IDE\IDE98\MSE\1033

> ! *Note that there may be more than one file named HTMLREF.CHM on your machine (don't you just love the concept of reusable file names?), so ensure that the file you have is actually the "*Internet Development SDK*".*

The *ExecCommand()* method executes a command on the current document, selection, or a given range. It returns a logical value indicating success or failure. The syntax is:

```
lResult = object.execCommand(cCmdName [, lShoUI] [, uParam])
Where:
    cCmdName (Mandatory) is any of the defined "Command Identifiers"
    lShoUi (Optional) flag to allow the display of any UI for the command
    uParam (Optional) any additional parameter required by the command
```

The "command identifiers" referred to are simply the names of the commands that the document object recognizes. The full list is documented in the Help file, but there are far more of them than are exposed through the *ExecWB()* method of the browser control. While most are concerned with accessing and modifying the content of the page, *SelectAll, Copy,* and *UnSelect* are supported. We could, therefore, have used *ExecCommand()* instead of *ExecWB()* in the *DoCopy()* method of our FRMBROW.SCX form to select the contents of the page and copy it to the clipboard. The necessary code is included, commented out, in the method. To run it, simply comment out the first block and uncomment the following:

```
*** Select all and copy to clipboard
*** Method [2] Using the Document Object
WITH ThisForm.pgfbrowser.Page2.oExplorer
  .Document.ExecCommand( 'SelectAll', .F. )
  .Document.ExecCommand( 'Copy', .F. )
  .Document.ExecCommand( 'UnSelect', .F. )
ENDWITH
```

As you can see, for something this simple, there is little benefit in accessing the document object directly, but it does give access to greater functionality than is supported by the browser control's interface.

Using the DOM (*Example: frmKbase.scx*)

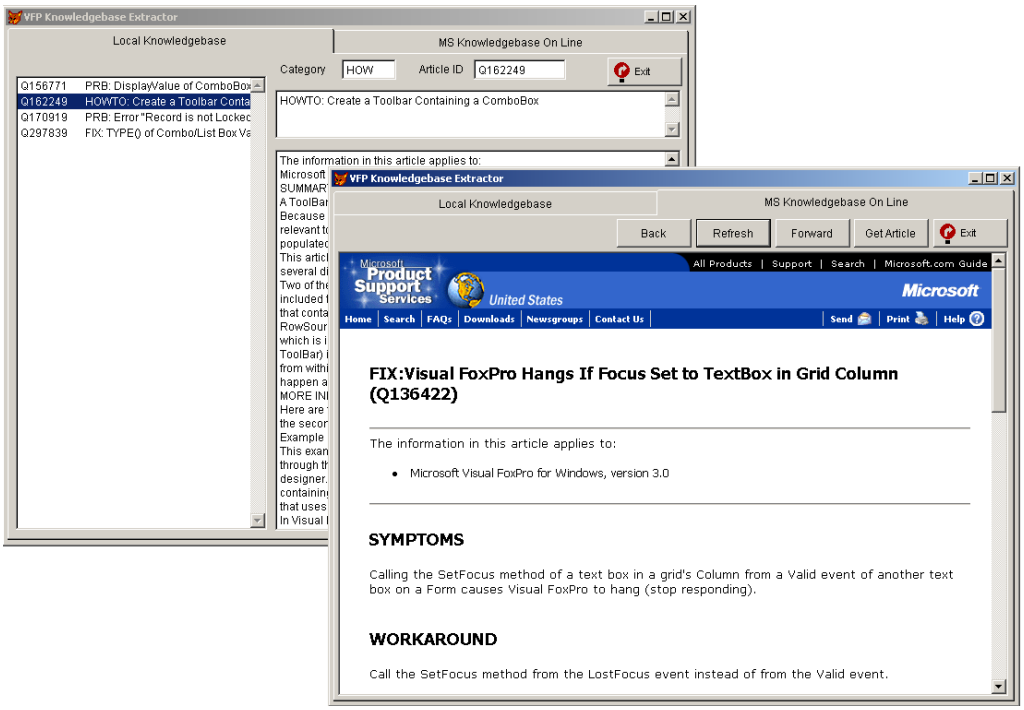


***Figure 5**. VFP Knowledgebase extractor (frmkbase.scx).*

To illustrate using the Document Object Model in a realistic environment, the example form, FRMKBASE.SCX, accesses the online Microsoft Knowledgebase and programmatically extracts the content from an article into a local Visual FoxPro table (see **Figure 5**). This example is based on an idea originally described by our friend and colleague, Remi Carron, in his article "Having Fun With Internet Explorer," which was first published in the *Dutch Developer's User Group Magazine*.

Our form uses two tables named KBASE.DBF and CATEG.DBF, whose structures are described in **Table 1**.

***Table 1**. Tables used by the Knowledgebase extractor.*

Field	Type	Description
KBASE.DBF		
KBKey	C 3	Foreign key into Category table
KBQNum	C 8	Knowledgebase Article "Q" number
KBTitle	C 150	Title of the article
KBBody	M 4	Article contents
CATEG.DBF		
KBKey	C 3	Category Key Code
kbCat	C 15	Category Description

The first page simply displays the available article numbers and titles in a list box, which uses the KBASE.DBF table as its *RowSource* so that no special code is needed to synchronize the list with associated display fields. That is all that this page is for, and we need pay no more attention to it. All the interesting stuff happens on page two.

When the second page is activated for the first time, the value, which is set in the form's custom *cDefaultURL* property, is used to bring up the Knowledgebase home page in the browser control. This page provides full search facilities so that we can locate an item of interest. When selecting an article from the result list, the default behavior is to open a separate Web browser window. In this case, we do not want that to happen so we have to ensure that, when selecting an item from the result list, we use the right-click menu and choose "Open". The selected article is then displayed in the form's browser.

Once we have an article selected, we can insert its contents into our table by clicking on the "Get Article" button. This calls the form's custom *GetArticle()* method, which is where all the work is done. The first thing that we do is to get a reference to the current document object and retrieve the document's title from the aptly named *Title* property.

```
WITH ThisForm.pgfkbase.page2.oExplorer
  *** Get the current document into a local variable
  loDoc  = .Document
  *** Retrieve the title from the document
  lcTitle = ALLTRIM( loDoc.Title )
```

Fortunately for us, Knowledgebase articles use a standard format for their URL that includes the article number as the final part of the address. Since the document object's *LocationURL* property always contains the currently displayed page's URL, we can use that to retrieve the "Q" number as follows:

```
  *** Extract the "Q" number
  lcQNum  = SUBSTR( loDoc.url, AT( ";Q", loDoc.url ) + 1 )
```

In order to select only the relevant part of the article's content, we need some way of detecting where it actually starts. (Remember, the "content" includes all of the text on the entire page.) The obvious thing to use is the title, but, unfortunately, we have found that the text used as the "*Title*" of the document is not always exactly the same as that which is embedded in the body of the article! If we happen to have chosen an article with a static prefix

(for instance, *PRB:* or *BUG:*), we can use that; otherwise, we just have to hope that, even if there is not an exact match, the first few characters are the same.

```
IF AT( ": ", lcTitle ) > 0
  *** We can use the prefix as the identifier
  lcPref  = LEFT( lcTitle, AT( ": ", lcTitle ) + 1 )
ELSE
  *** We have to hope the first 10 chars on the title in the text
  *** are actually the same as the document title - this is NOT always true!!
  lcPref  = LEFT( lcTitle, 10 )
ENDIF
```

We have now got all that we need from the Document object; next we must drill down into it to retrieve its Body object—which is where the contents of the page are held. The Body object stores the page as plain text in its *InnerText* property, and as HTML in its *InnerHTML* property. Since we are going to store the data in a Visual FoxPro table, we only want the plain text. In order to handle it more easily, we use the **ALINES()** function to get the document's contents directly into an array:

```
*** Now get a reference to the document body object
loBod  = loDoc.body
*** Get the text from the body into an array for parsing
lnLines = ALINES( laText, loBod.InnerText )
*** And initialise a couple of variables
llWriteOut = .F.
lcOutPut = ""
```

All that is left to do is to loop through the array and select the lines of text that we want. We have chosen to take only text that appears between the document title and the footer region (the footer region begins with the date published). The lines we select are added to the variable named *lcOutPut*, adding back the carriage return and line feed characters as necessary.

```
*** The following block of code parses the entire array, but we only want to
*** write certain lines out to the final output. The llWriteOut Flag is used
*** to control when we start writing data out.
FOR lnCnt = 1 TO lnLines
  *** Get the line, preserve leading spaces
  lcLine = RTRIM( laText[ lnCnt ] )

  *** Have we found the start point yet?
  IF NOT llWriteOut
    *** Is this the starting line we want
    IF lcLine = lcPref
      *** Start from here
      llWriteOut = .T.
    ELSE
      *** Keep trying
      LOOP
    ENDIF
  ENDIF

  *** If we get here, we must have started writing data out, so the
  *** Question now is, have we reached the end?
  IF "Published" $ lcLine AND "Issue" $ lcLine
```

```
      *** Yes, we have reached the end of the text we want, so get out
      EXIT
    ENDIF

    *** If we get this far, we must want this line of text
    *** So just add it to the output string
    *** Note that ALINES() removes CRLF, so we must add them back
    lcOutPut = lcOutPut + lcLine + CHR(13)+CHR(10)

  NEXT
```

Finally, we call a simple pop-up form (FRMGETCAT.SCX) to assign the article to a category and write the data to the table.

```
  *** Get the category designation
  lcCateg = ""
  DO FORM frmGetCat TO lcCateg
  *** And Write the data out
  IF NOT SEEK( lcQNum, 'kbase', 'kbqnum' )
    INSERT INTO kbase VALUES (lcCateg, lcQNum, lcTitle, lcOutPut )
  ELSE
    MESSAGEBOX( "You already have that article in your database", 16, ;
                "Not needed!" )
  ENDIF
ENDWITH
RETURN
```

As you can see, there are a lot of assumptions in this code, but it does work providing that you choose articles that have "Q" numbers. Clearly this example could easily be made much more generic—but we have left that task as an exercise for the reader. Documentation for the DOM can be found in the "Internet Development SDK" (HTMLREF.CHM).

How do I create a hyperlink in a VFP form? (*Example: frmHl01.scx*)

As with so many things in Visual FoxPro, there are several ways of doing this. The simplest is to instantiate an object based on the Visual FoxPro *Hyperlink* base class. This class exposes a method named *NavigateTo()* that can accept a URL and, when executed, opens an instance of Internet Explorer and navigates to the specified location. This is demonstrated in the example form that uses three different controls to initiate the hyperlink (see **Figure 6**).

The hyperlink object is created in the Form's *Init()* method using the *AddObject()* method so that the hyperlink is created as child object contained within the form. The benefit of this approach is that we do not need to worry about leaving dangling references; the object will be destroyed when the form is released:

```
*** Instantiate the Hyperlink Object
ThisForm.AddObject( 'oHlink', 'hyperlink' )
```

The image, button, and two hyperlink labels all call the same custom *Navigate()* method on the form, passing the required destination as a parameter. Notice that the label and command button have both been set up to mimic the standard behavior of a hyperlink; the color changes from blue to mauve when the link is executed. Also note that we have used the new *MouseEnter()* event to change the mouse cursor pointer to a hand when over the link.

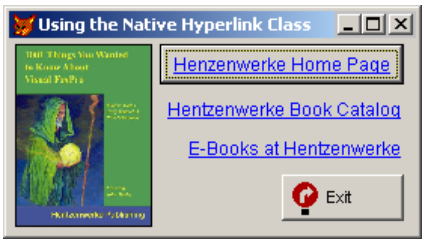


***Figure 6**. Using the hyperlink base class (frmhl01.scx).*

The form's *Navigate()* method merely calls the hyperlink object's *NavigateTo()* method and passes the URL:

```
LPARAMETERS tcURL
*** Use the hyperlink object o navigate to the page
ThisForm.oHLink.NavigateTo( tcURL )
```

This is indeed simple, but there are a couple of fairly serious limitations with this particular base class. First, it was designed to operate in the context of the Visual FoxPro "Active Document" interface, and it will only navigate to a URL or a supported document type. If you attempt to navigate to something that is not recognized as "*supported,*" nothing happens. There is no error; the *NavigateTo()* method is simply not executed.

Second, as you may already have noticed, clicking on any object that triggers the link always opens a new instance of the browser. This is because, when used in a plain form (or program) rather than in an active document, the class determines what application is defined as supporting the active document interface (for instance, Internet Explorer) and then starts a new instance of that application. You have no control over this behavior; that is simply what it does.

Finally, the help file on the hyperlink base class includes a note that:

> *The Hyperlink object is supported only in Microsoft Internet Explorer. Use the Visual FoxPro Hyperlink Button, Hyperlink Image, or Hyperlink Label foundation classes in the _Hyperlink class library for browser independent navigational capabilities.*

The conclusion is that, while the hyperlink base class is indeed very simple to use, it is also rather simplistic. Unless all you need is a simple one-time link, it looks like a better approach is required.

What about the FoxPro foundation classes?

The Visual FoxPro foundation classes include a class library named _HYPERLINK.VCX in which the *_Hyperlinkbase* class provides a wrapper around the base class to give a little more functionality. This class is then reused in the definition of three additional classes, one each for a command button, label, and image, which provide mechanisms for implementing hyperlinks in Visual FoxPro forms.

All three of these classes use the same interface and methodology. They each instantiate an instance of the hyperlink wrapper class and expose four properties that are used to control behavior:

- *lNewWindow* Specifies whether a new instance of the browser should be opened. Default = False.
- *cTarget* The target location to which the link leads (either a URL or a document). Expects a character string.
- *cLocation* The specific location within the target document to jump to. If omitted, the default document is assumed.
- *cFrame* The name of the frame within the target document to jump to. If omitted, the default document is assumed.

If explicit control is needed, the FoxPro Foundation classes do provide that functionality. When instantiated, each creates an instance of the *_HyperLinkBase* class and exposes a method named *Follow()*. This method first sets the *lNewWindow* property on the hyperlink object to be the same as its own property, and then calls its *NavigateTo()* method, passing the contents of the other three properties.

Unfortunately, as all too often is the case with the foundation classes, there are problems with trying to use them in production code. First, you need to ensure that if you change their locations *all* the necessary relative paths are updated, or you'll find yourself with a series of "*unable to locate*" errors. Second, you cannot simply abstract just one library—there are dependencies on other class libraries and objects. Third, like most of the foundation classes, these classes are not well documented and their code is almost entirely devoid of comments. So without a lot of effort it is hard to be certain of exactly what they are doing (or even why they are doing it). However, they appear to work in their own environment, with Internet Explorer, as evidenced by the sample form HYPERLNK.SCX that ships with Visual FoxPro. We have not tested them independently, or tried to use them with any other browser.

Creating your own hyperlink classes (*Example: frmHl02.scx*)

In fact, when we started to look into this topic in more detail we quickly realized that the best way to implement a browser-independent hyperlink class was not to try and utilize the hyperlink base class at all. Instead we turned our attention to the Windows API in general, and the **`SHELLEXECUTE()`** function in particular. In general terms, this function executes a specified action (referred to as a "verb") on the specified item. In this particular context we are only interested in the "*Open*" verb. When asked to "open" an item, **`SHELLEXECUTE()`** determines the type of item it has been given, locates the application that is associated with that type, and, if it is not already open, launches the application. Once the application is open, the item is simply passed to that application for processing.

The result is that if you pass **`SHELLEXECUTE()`** a URL, it will launch whichever browser is defined as default, and allow it to navigate to the URL. The result is identical to using the hyperlink base class, but we are no longer limited to using Internet Explorer. More importantly, nor are we bound by the restrictions imposed by the Active Document interface. Providing that an association between a file type and an installed application exists, we can navigate to any valid file or location.

The class library, CH05.VCX, includes three classes, one each for a label, command button, and image, which implement a common interface as shown in **Table 2**.

***Table 2**. Hyperlink class interface.*

Name	Type	Description
cJumpTarget	Property	Used to specify the target file or location.
DoJump	Method	Exposed method that executes a jump. Accepts a target location as a parameter; if nothing is passed, uses the contents of the *cJumpTarget* property.
Initialize	Method	Protected method, called from the Init(), which declares the **SHELLEXECUTE()** function.
JumpTo	Method	Protected method that calls the **SHELLEXECUTE()** function with the "Open" verb. Returns the numeric result of the function.

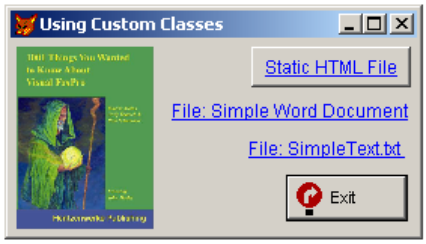


***Figure 7**. Using the hyperlink custom classes (frmhl02.scx).*

Note that you can drop these classes onto a form, and set their *cJumpTarget* property at design time (see **Figure 7**). Clicking on the instance immediately executes the jump to whatever is specified by the property. Alternatively, you can instantiate any of them in code and execute a jump by calling the *DoJump()* method and passing the required target as a parameter. The following code opens Windows Explorer to browse the Visual FoxPro home directory:

```
oLink = NEWOBJECT( 'xhyperlabel', 'CH05.vcx' )
oLink.DoJump( FULLPATH( HOME()))
```

You will notice, if you experiment with these classes, that we still have no direct control over exactly how an application displays the result of a jump. This is not surprising when you remember that the default behavior of **SHELLEXECUTE()** is that if the required application is not already running it is opened. Once an instance of the application is open, the target file is displayed by whatever method that application defines. For instance, if Internet Explorer is your default browser, navigating to a URL, or opening an HTML or XML file, will always occur within the currently open window, replacing any existing content. However, if the target is a Microsoft Word document, Word will always open that document in a new window, without closing any existing document windows.

However, this seems, to us, less important than the ability to be independent of any particular browser, or interface.

How do I use Web Services in my applications?

Before we get down to the specifics of "*how*," we should first provide some background for those who are unfamiliar with Web Services. A Web Service is defined as:

Programmable application logic accessible using standard Internet protocols

In fact, it is simply an entity (usually a class) that provides some sort of functionality, but that makes itself accessible to any system that is capable of communicating using XML and HTTP. The result is that, by exposing an object as a Web Service, we can avoid all the incompatibility issues that arise when applications written in different languages attempt to interact with each other. The basis is that each entity is responsible for describing itself, and its interface, in a WSDL (Web Service Description Language) file that is published on the Internet.

When access to the entity is required in an application, this standardized definition is retrieved by software on the client system and used to create a local proxy for the entity (the "SOAP client") that binds all the methods described in the WSDL to itself during initialization. Thereafter all communication between an application and a Web Service is routed through this object, which essentially acts as a two-way interpreter. The actual communication consists of SOAP (Simple Object Access Protocol) messages, which use formatted XML. The SOAP client provides a high-level API that wraps the various components needed to create and interpret the messages (see **Figure 8**). For more details on SOAP, its implementation and capabilities, see the SOAP Developer Resources Web site (http://msdn.microsoft.com/soap/).

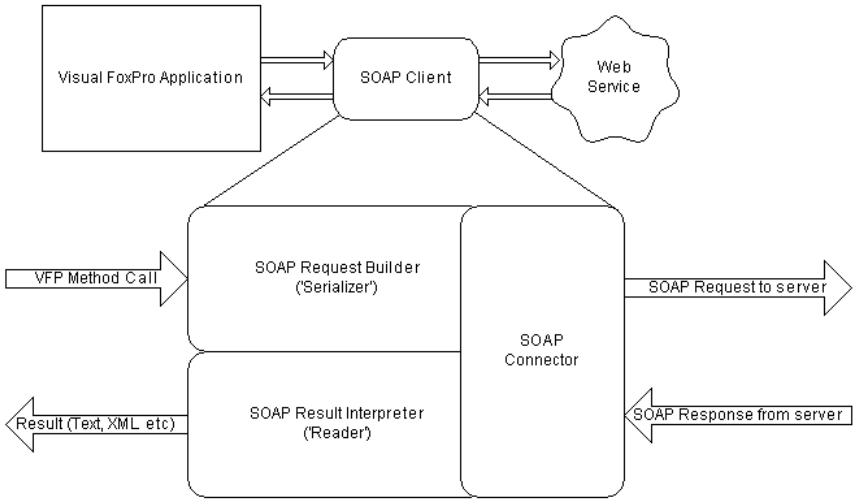


***Figure 8**. Simplified SOAP data flow.*

Visual FoxPro Version 7.0 provides native support for Web Services by implementing a set of extensions to the Microsoft SOAP Toolkit 2.0. These classes integrate registering Web

Services into its IntelliSense engine, and define wizards that simplify the task of creating and publishing Web Services. They are stored in the foundation class library, _WEBSERVICES.VCX.

Note that the SOAP Toolkit *must* be installed in order to use Web Services from within Visual FoxPro, but, although a distributable copy of the toolkit ships with Version 7.0, it is not installed automatically. The Help file that ships with the toolkit details the requirements for creating distributable applications that are Web-Service enabled. (The toolkit, and latest service releases, are available for download, free of charge from the Microsoft MSDN Library Web site at http://msdn.microsoft.com/library/.)

There are, therefore, two ways of accessing Web Services in your applications. First, we can use the Visual FoxPro extensions and let the IntelliSense engine create the necessary code for us. Second, we can use the SOAP Toolkit API directly in our code. Of course, before we can access a Web Service, we have to locate one and find its WSDL file. For Web sites that provide access to their functionality through Web Services, the WSDL is usually accessible directly (for example, the FoxCentral.net home page includes a link to the "Web Services Documentation" page). Alternatively there are sites that provide lists of available Web Services, together with the information needed to access them. One of the best known of these is XMethods at www.xmethods.net/.

The list of available Web Services is growing and changing all the time. Although correct at the time of writing, we cannot guarantee that all the services used to illustrate the following section will still be available or that, even if they are, they will not have changed significantly.

How do I register a Web Service using the VFP extensions?

The Web Service registration interface can be accessed through the Types tab of the IntelliSense Manager. A button on that tab, labeled "Web Services…", brings up the registration page (see **Figure 9**). Simply type a descriptive name for the service that you want to register into the "Web Service Name" combo box and enter (or preferably paste) the WSDL file location into the "WSDL URL Location" combo box. When done, click the Register button to initiate the registration process.

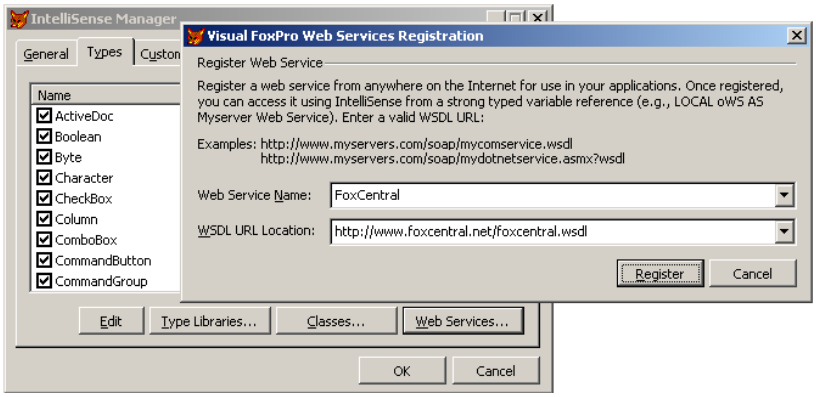


***Figure 9**. Registering a Web Service using the IntelliSense Manager.*

Note that this dialog can also be invoked programmatically using:

```
DO (_wizard) WITH "project",,"Web","IntelliSense"
```

All that seems to happen is that after a few moments a message box appears saying "Finished generating IntelliSense scripts correctly". This may not look like much, but Visual FoxPro has actually been quite busy behind the scenes.

First, a new record has been inserted into your FOXCODE.DBF table (this is a "T" type record; see Chapter 3 for details). Next, the specified location was interrogated, and the details of the WSDL file were retrieved and interpreted. Finally, a record has been added to another table, named FOXWS.DBF, which stores the information from the WSDL file. This table is created on the fly if it does not already exist, in the same location as your FOXCODE.DBF table. It is used to store information about existing Web Services that you register, and also for recording the details of those that you create yourself in Visual FoxPro. In the context of registering existing Web Services, only eight of its fields are used, as follows:

Field	Used for
type	Type identifier
name	Name that was specified when the service was registered
menu	List of available methods for the service
tips	Calling prototypes for available methods
uri	Location of the WSDL file for the service
class	Name of the server port that handles SOAP messages
timestamp	Date and time the record was created
uniqueid	Generated ID for the record

To make use of the newly registered Web Service, create a new program file and enter a **LOCAL** variable definition using the new **AS** clause. IntelliSense pops up a list of registered items that includes Web Service names (see **Figure 10**). Selecting the name of a Web Service fires off an IntelliSense script that generates the code shown in **Figure 11**.

To all intents and purposes the object referenced by "*loWS*" behaves as if it were just another local Visual FoxPro object, and IntelliSense shows its methods and provides associated tips for parameters (this information is, of course, gleaned from the WSDL file).

Note that there is, in Version 7.0, no provision to delete the reference to a Web Service once it has been registered (which seems rather a strange oversight). You can, of course, simply open the FOXWS.DBF, delete the record, and then pack the table. You also have to delete the entry from FOXCODE.DBF (and clean up the table); otherwise, the service name will continue to appear as an available IntelliSense item.

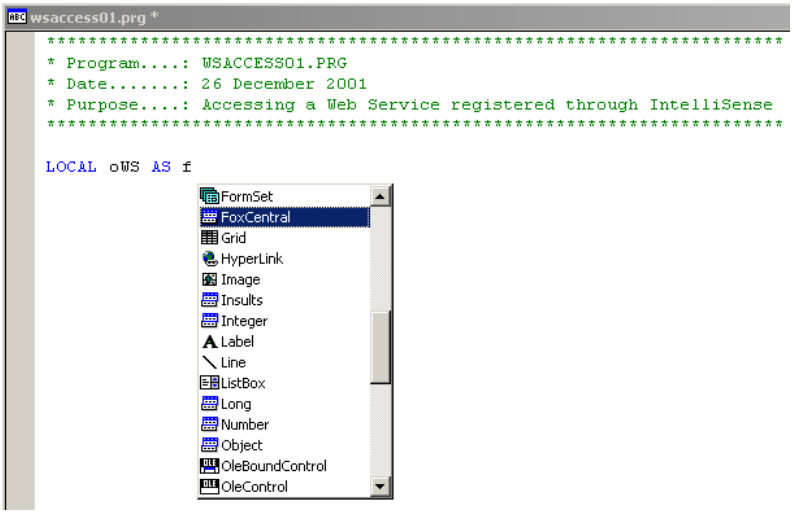


***Figure 10**. Creating the local definition to access a Web Service.*

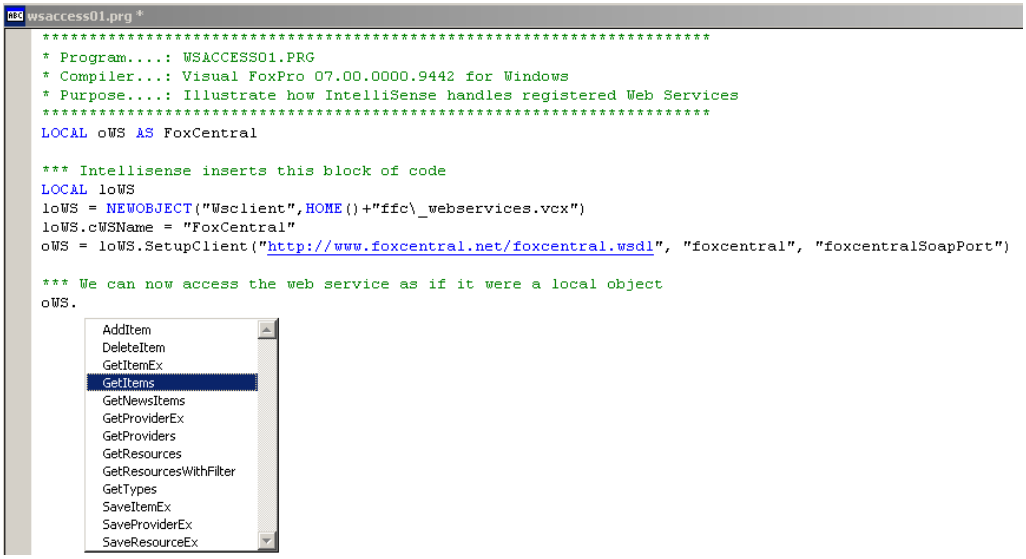


***Figure 11**. Creating the local definition to access a Web Service.*

How do I use a registered Web Service? (*Example: frmWs01.scx*)

This is rather like asking, “How long is a piece of string?” Web Services can be created to do almost anything you wish; a glance at XMethods.com revealed Web Services available for:

- Calculating the optimum number of lights on a Christmas tree
- Retrieving nucleotide sequences and associated information
- Locating music teachers by ZIP Code
- Checking the current bid price of an eBay auction

and many, many others. How you use one depends largely upon what functionality it exposes, and how you choose to implement that functionality in your application. Remember that a Web Service does not have any user interface of its own.

The example for this discussion uses the Web Service provided by FoxCentral.net to provide the functionality to a VFP form that allows you to interrogate and retrieve filtered lists of items posted to the site. The SOAP client is initialized in the form's custom *SetForm()* method, which is called directly from *Load()*:

```
LOCAL lcXMLStr
WITH ThisForm
  WAIT 'Connecting to Web Service....' WINDOW NOWAIT
  *** Initialize the SOAP client and connect to service
  .oWS = NEWOBJECT("Wsclient",HOME()+"ffc\_webservices.vcx")
  .oWS.cWSName = "FoxCentral"
  .ows = .oWS.SetupClient("http://www.foxcentral.net/foxcentral.wsdl", ;
                                    "foxcentral", "foxcentralSoapPort")
```

Next we use methods provided by the Web Service to get the lists of content providers (*GetProviders()*) and Message Types (*GetTypes()*) into local cursors (see **Figure 12**). The data from this Web Service is returned as XML, so we just use the `XMLTOCURSOR()` function to create the cursors:

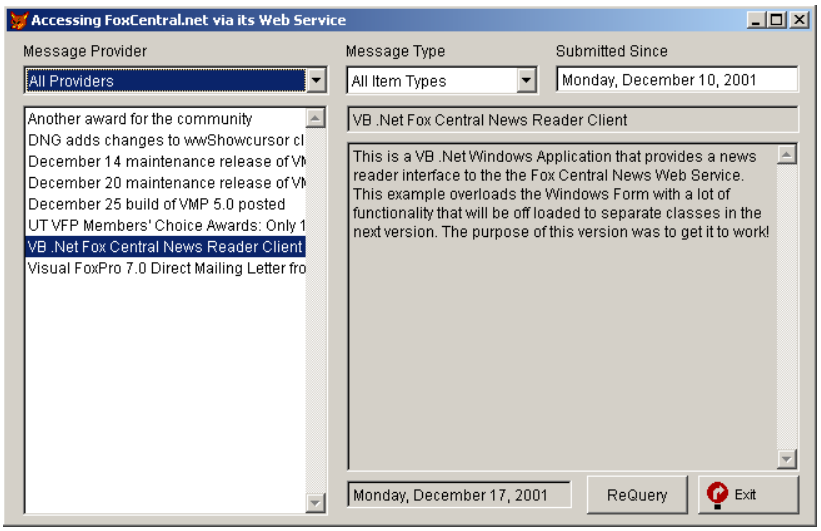


***Figure 12**. The FoxCentral Web Service in a form (frmwso1.scx).*

```
  *** Get the list of Providers
  lcXMLStr = ""
  lcXMLStr = .oWS.GetProviders()
  IF NOT EMPTY( lcXMLStr )
    XMLTOCURSOR( lcXMLStr, 'curProvs' )
  ELSE
    *** Nothing back!
    MESSAGEBOX( 'Unable to retrieve Providers List from FoxCentral', ;
                                 16, 'Cannot Initialize Form' )
    RETURN .F.
  ENDIF
  *** Add the ALL entry and index on ID
  INSERT INTO curProvs (pk, company) VALUES (0, 'All Providers' )
  INDEX ON company TAG company

  *** Get the list of Item Types
  lcXMLStr = ""
  lcXMLStr = .oWS.GetTypes()
  IF NOT EMPTY( lcXMLStr )
    XMLTOCURSOR( lcXMLStr, 'curTypes' )
  ELSE
    *** Nothing back!
    MESSAGEBOX( 'Unable to retrieve Message Types from FoxCentral', ;
                                16, 'Cannot Initialize Form' )
    RETURN .F.
  ENDIF
  *** Add the ALL entry and index on ID
  INSERT INTO curTypes (type, description) VALUES ( 'All', 'All Item Types' )
  INDEX ON description TAG desc
```

Finally we create the Items cursor. This cursor is going to be re-queried so, rather than allowing it to be closed each time we send a request to the server, we are using the "*safe select*" approach (which is handled in the form's *GetItems()* method), but this requires that we create the cursor's structure explicitly.

```
  *** Create the Items Cursor explicitly
  CREATE CURSOR curitems ( ;
    SUBJECT         C (100,0 ), ;
    CONTENT         M (  4,0 ), ;
    LINK            M (  4,0 ), ;
    XMLLINK         M (  4,0 ), ;
    SUBMITTED       T (  8,0 ), ;
    PRIVATE         N (  1,0 ), ;
    IMAGELINK       M (  4,0 ), ;
    MODE            I (  4,0 ), ;
    COMPANY         C (100,0 ), ;
    COMPANYWEBSITE  M (  4,0 ), ;
    PK              I (  4,0 ), ;
    PROVIDERPK      I (  4,0 ))
  *** No buffering here, it'll be read only
  CURSORSETPROP( "Buffering", 1, 'curItems')
  INDEX ON subject TAG subject

  *** Populate the Items Cursor with default values
  .GetItems()
ENDWITH
```

The *GetItems()* method is called by both the *SetForm()* and the *UpdateDisp()* methods. It simply initiates the request for data from the server by calling the SOAP client's *GetItems()* method, passing the parameters, which are:

- The cut-off date to apply to queries in either *Date* or *DateTime* format. Defaults to 10 days before today's date.
- The time zone to apply. Defaults to zero (*GMT*).
- The Primary Key of the content provider whose content is required. Defaults to zero (*All providers*).
- The Type of message that is required. Defaults to "*All*".

The code is quite straightforward, with the possible exception of the safe select technique, which may not be familiar to everyone:

```
LPARAMETERS tdCutOff, tnTimeZone, tnProvPK, tcType
LOCAL ldCOff, lnZone, lnProv, lcType, lcXMLStr

*** If nothing passed, default to 10 days ago
ldCOff = IIF( INLIST( VARTYPE( tdCutOff), "D", "T") ;
              AND NOT EMPTY( tdCutOff ), ;
              tdCutOff, DATE()-10 )
*** If nothing passed default to TimeZone 0
lnZone = IIF( VARTYPE( tnTimeZone ) = "N" ;
              AND NOT EMPTY( tnTimeZone ), ;
              tnTimeZone, 0 )
*** If nothing passed default to all Providers
lnProv = IIF( VARTYPE( tnProvPK ) = "N" ;
              AND NOT EMPTY( tnProvPK ), ;
              tnProvPK, 0 )
*** If nothing passed default to all Message Types
lcType = IIF( VARTYPE( tcType ) = "C" ;
              AND NOT EMPTY( tcType ), ;
              tcType, 'All' )

*** Use a "safe select" to ensure the target cursor remains open
ZAP IN curItems

*** Now try and get the requested data as XML
lcXMLStr = ""
lcXMLStr = ThisForm.oWS.GetItems( ldCOff, lnZone, lnProv, lcType )
IF NOT EMPTY( lcXMLStr )
  *** We got something back, so convert to a transient cursor
  XMLTOCURSOR( lcXMLStr, 'curtemp' )
  *** And populate the 'real' target cursor using APPEND
  SELECT curItems
  APPEND FROM DBF( 'curtemp' )
  USE IN curTemp
ENDIF
```

If we were to run the `XMLTOCURSOR()` function directly to the target cursor (*curItems*), FoxPro would close and re-create the cursor each time the function was called. This would have consequences in three ways; first we would lose any specific buffering that we had

established, and second we would lose any indexes that had been created for the cursor. Third, and most seriously, if the cursor were being used as the *RecordSource* for a grid, closing the cursor would cause the grid to re-initialize itself and lose its custom settings.

By running the XML into a temporary cursor and using the **ZAP** and **APPEND** commands to clear and re-populate the target cursor, we avoid all of those issues.

Note that although this example shows only how to retrieve data from a Web Service, it is perfectly possible to send data to a Web Service, providing that it has the necessary functionality. The example used here, FoxCentral.net, does in fact allow registered providers to submit items using the Web Service.

How do I find out how to use a Web Service? (*Example: frmWs02.scx*)

There are at least three ways to do this. First, the originators of a service usually provide some information about the exposed methods—their parameters, return values, and so on. Often they will provide examples too—though these are rarely written in Visual FoxPro. However, not all sites are so helpful, and even if they are, the particular item of information that you want may not have been included.

Second, you can simply attempt to register the WSDL using the IntelliSense engine. Once registered you can either inspect the entry in the FOXWS.DBF table or use the interactive lists provided by IntelliSense to determine the information. The only snag with this, as we mentioned earlier, is that there is no simple way to delete an item once you have registered it. If what you are trying to decide is whether you want to register it or not, this is not a good option.

The last possibility is to read the WSDL file and see what it has to say. The whole purpose of the WSDL is to *describe* the Web Service, after all. Here is a small extract from the WSDL file, created by Ed Leafe, that defines his Web Service for searching the "ProFox" message archives.

```
<message name="ProFoxWS.GetMsg">
  <part name="MsgNum" type="xsd:int" />
</message>
<message name="ProFoxWS.GetMsgResponse">
  <part name="Result" type="xsd:string" />
</message>
<message name="ProFoxWS.SearchTextLinks">
  <part name="Text" type="xsd:string" />
  <part name="Begin" type="xsd:dateTime" />
  <part name="End" type="xsd:dateTime" />
  <part name="CaseSensitive" type="xsd:boolean" />
</message>
<message name="ProFoxWS.SearchTextLinksResponse">
  <part name="Result" type="xsd:string" />
</message>
```

It is worth noting at this point that Ed's SOAP server was not written using VFP, or any other Microsoft product, and is actually running on a Mac platform. Yet a VFP application running under Windows with an Internet connection can instantiate this class, and call its methods, as if it were a native object. This is precisely why Web Services are so useful and why they are likely to become an increasingly common component of applications in the future.

However, we have to say that we do not really find it very easy to read raw XML like this, especially when the file is large. Besides, what exactly do all of the tags mean? The SOAP Toolkit includes a class that can be used to read a WSDL file and return meaningful information for us.

An overview of the SOAP Toolkit

The Microsoft SOAP Toolkit is installed, by default, at:

```
C:\Program Files\Common Files\MSSoap\Binaries\MSSOAP1.dll
```

It can be viewed in the Object Browser by opening this file. There is, unfortunately, no Help file supplied with this version of the toolkit, but there are both documentation and technical white papers for SOAP available online from http://msdn.microsoft.com/products/ (or as part of the MSDN Universal Subscription) and we have included, in the sample code for this chapter, an extract of the constant definitions for the SOAP Toolkit (SOAPCONST.H). In fact, the SOAP Toolkit includes a total of 17 classes (see **Figure 13**), but a full discussion of them all is beyond the scope of this chapter.

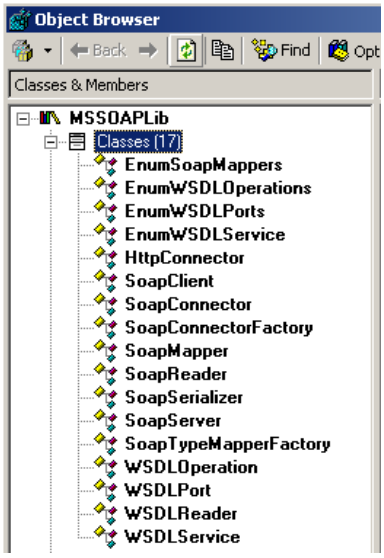


***Figure 13**. SOAP Toolkit (V2.0) classes.*

For the purposes of this section we are only interested in the *WSDLReader* class. It provides the necessary methods that allow us to access and interpret WSDL files without having to deal directly with the raw XML. However, you will note, when you examine the *WSDLReader* class in the Object Browser (see **Figure 14**) that it does not support the *IDispatch* interface. This means that is not an Automation server and so cannot be instantiated in Visual FoxPro using `CREATEOBJECT()`. Instead, we have to use `CREATEOBJECTEX()`, which, although described in the Help file as follows:

> *Creates an instance of a registered COM object (such as a Visual FoxPro Automation server) on a remote computer*

actually creates a local instance when called like this:

```
oReader = CREATEOBJECTEX( "MSSOAP.WSDLReader", "", "" )
```

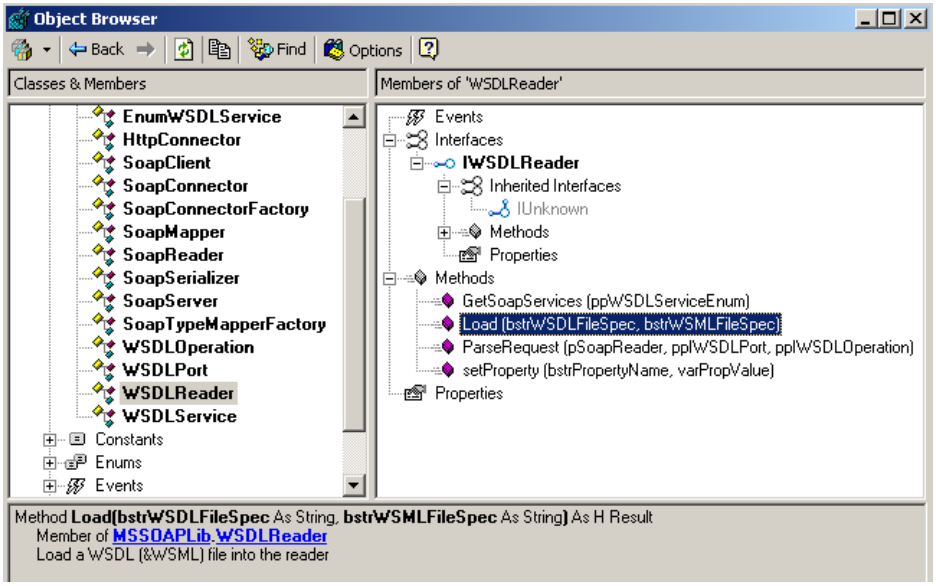


***Figure 14**. WSDLReader class interfaces and methods.*

The sample form (see **Figure 15**) shows how we can use the *WSDLReader* class to extract information from a WSDL file.

The WSDL Inspector form (*Example: frmWs02.scx*)

The WSDL Inspector is designed to show how to use the SOAP Toolkit to retrieve information from a WSDL file. In order to make sense of it, we have to remember that a Web Service is defined hierarchically. A "Web Service" actually consists of one or more "*Services*," which you can think of as being analogous to "*class libraries*". Each *Service* exposes one or more "*Ports*," which, continuing the analogy, equate to "*classes*". A *Port*

exposes one or more "*Operations*," which are equivalent to the methods of a class, and, finally, each Operation has one or more "*Parts*" that define the parameters and return value.

To see how the form works, type (or paste) the URL for a WSDL file into the form's location text box.

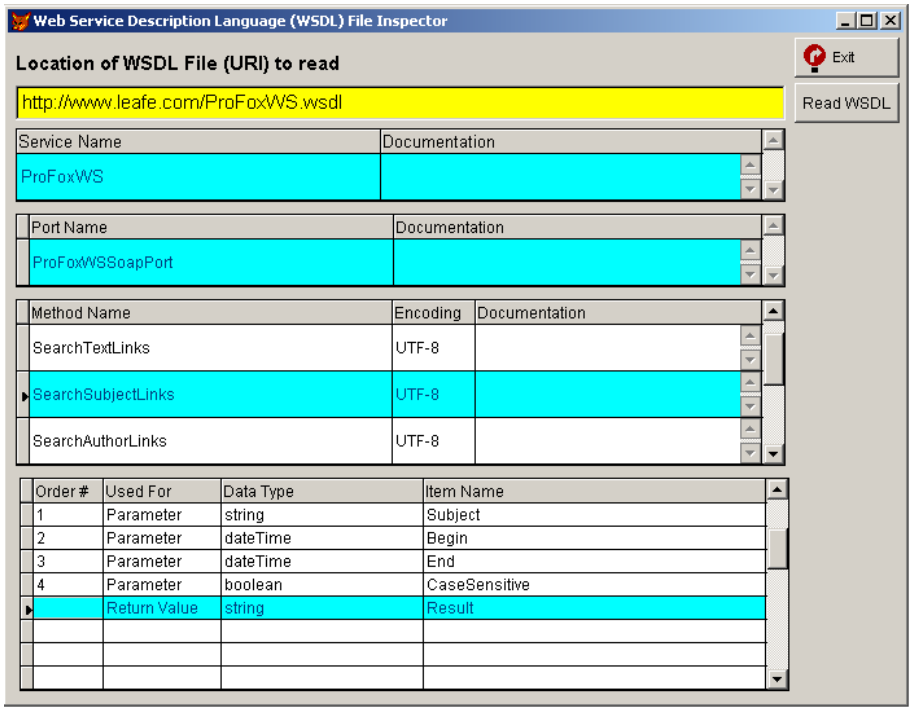


***Figure 15**. The WSDL Inspector form (frmws02.scx).*

The URL of a WSDL file is usually referred to as a "Uniform Resource Identifier" (URI). The reason is that the term URI covers both URLs (which specify the location of a resource) and URNs (which specify the identity of a resource rather than its location).

To make things easier, the file wsdl.txt, included with the sample code, contains the URIs for Ed Leafe's ProFox service, the FoxCentral.net service, the xMethods Listing service, and a Session State Store service. Having supplied the URI, click the "Read WSDL" button to populate the grids and display the basic information for each of the four levels of the WSDL file hierarchy. Figure 15 shows the results obtained from the ProFox WSDL file.

Note that this form does not actually save anything to permanent storage. The IntelliSense registration process already deals adequately with that. The intention here is to provide a simple way to check out a Web Service *without* having to register it. Of course, extending this form to include retrieving a WSDL file and saving the details would not be difficult, and so we have left that as an exercise for the reader.

The *Load()* method of the form merely creates the four cursors that are used to hold the information gleaned from the WSDL file. A separate cursor is used for each of the four levels (*Service*, *Port*, *Operation*, and *Part*) so that we can easily handle the one-to-many relationships. The real work is handled by the custom *ReadWSDL()* method, which controls the processing. Having checked that a character string has been entered, the method first creates an instance of the WSDLReader and attempts to load the specified file. If that succeeds, it clears the local cursors and initiates the process of drilling down through the file hierarchy, populating the relevant cursors at each level. After updating the form, the WSDLReader object is released.

Instantiating the reader, and loading the WSDL file, is handled by the custom *LoadWSDL()* method, which expects to receive the URI for the WSDL file as a parameter. First it attempts to create the reader object, and assign it to a form property.

```
LPARAMETERS tcWSDL
LOCAL llFailed, lcErrWas
WITH ThisForm
  *** Create the Reader object if not already there
  IF VARTYPE( .oWSDLReader ) # "O"
    .oWSDLReader = CREATEOBJECTEX( "MSSOAP.WSDLReader", "", "" )
    IF VARTYPE( .oWSDLReader ) # "O"
      MESSAGEBOX( "Cannot instantiate WSDL Reader", 16, "Major Problem" )
      RETURN .F.
    ENDIF
  ENDIF
```

If the reader is created, we now try to load the WSDL file. There are three points to note about the code here. First, although the reader's *Load()* method actually accepts two parameters, the second parameter is for the *Web Services Meta Language* (*WSML*) file. This file is used on the server to map the operations exposed by the Web Service to specific methods of the COM object. It is purely a server-side component and is not necessary when simply reading a WSDL file. By default an empty string is passed when this parameter is omitted.

Second, the *Load()* method does not actually return a value; however, if it fails it will generate an error. There are actually a number of possible errors here, but all we are interested in at this point is whether the load succeeds or not so we wrap the call to the reader's *Load()* method in a localized error handler that simply returns True if the load fails.

```
*** Now try and load the WSDL file into the reader object
  lcErrWas = ON( "ERROR" )
  ON ERROR llFailed = .T.
  ** Try the load here
  .oWSDLReader.Load( tcWSDL )
  *** If it failed
  IF llFailed
    *** Try forcing an extension...
    lcWSDL = FORCEEXT( tcWSDL, 'wsdl' )
    *** And try again
    llFailed = .F.
    .oWSDLReader.Load( tcWSDL )
    IF llFailed
*** It really failed this time!
      MESSAGEBOX( "Unable to load the WSDL File", 16, "Load failed" )
```

```
  ENDIF
ENDIF
*** Restore the error handler and exit
ON ERROR &lcErrWas
RETURN NOT llFailed
```

Finally, note that although the usual format for a URI includes a "*.wsdl*" extension, this is not mandatory and it is possible to have a valid URI that omits it (for instance, the Session State Store service in WSDL.TXT). For this reason we first try to load the specified URI "as is" and, if it fails, re-try the load after forcing the extension. We mention this because, if you examine the code in the foundation class *_WebService.AddFoxCode()* method, you will note that the "*.wsdl*" extension is always forced before trying to load the file. As far as we can tell, it makes little difference; the reader appears to be able to resolve the URI whether the extension is there or not.

If the WSDL file is loaded successfully, we can initiate the process of reading it. By the way, you may have noticed in the Object Browser that the *WSDLReader* class exposes only a single "get" method (see Figure 14), but we keep talking about drilling down through a four-level hierarchical structure. The reason that there are no additional methods is that the reader's *GetSoapServices()* method returns an instance of the *EnumWSDLService* class. This consists of a collection of objects based on the *WSDLService* class (one object for each service) and has a *Next()* method that allows us to iterate through its collection. The *WSDLService* class defines a *GetSoapPorts()* method, which returns a collection of ports. The entire sequence is illustrated in **Figure 16**.

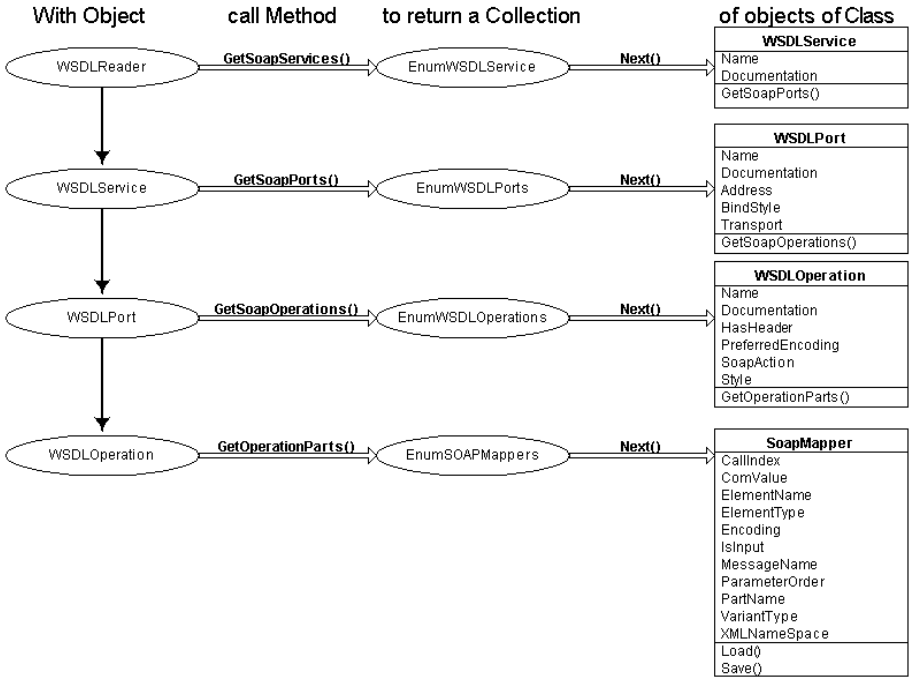


***Figure 16**. Reading a WSDL file using the SOAP Toolkit.*

The form's custom *GetServices()* method starts the process off. First we need to initialize two variables, one (*loSvcObj*) to receive the object, which is itself a collection of objects, returned by the call to the *GetSoapServices()* method. The other (*loCurSvc*) will be used to hold a reference to the current service as we work through the collection.

> ! *When working with these objects, they must always be initialized to 0. If you try to use the normal Visual FoxPro method, and assign them a* ***.NULL.****, you will get an error.*

```
LOCAL loSvcObj, loCurSvc, lnSvcPK, lcSvc, lcDoc
WITH ThisForm
    *** The reader expects these objects to be initialized to 0
    STORE 0 TO loSvcObj, lnSvcPK
    .oWSDLReader.GetSoapServices( @loSvcObj )
```

Having retrieved our collection of service objects, we iterate through it by calling its *Next()* method. We *must* pass all three parameters, which, according to the documentation, are:

- *The number of services to retrieve*. Must be 1. (It is not immediately obvious why this has to be passed because the only value that is allowed is "1" anyway. Maybe it's to allow for enhancements in the future)
- *The reference to be used for the service object*. This must be initialized to 0 otherwise an error occurs
- *The number of service objects retrieved*. This can be either 0 or 1 and, as far as we can tell, it doesn't matter which is passed (this too looks like "future-proofing")

```
    *** Iterate through the service object collection object
    DO WHILE .T.
      *** Get the next service Object
      loCurSvc = 0
      loSvcObj.Next( 1, @loCurSvc, 0 )
      IF VARTYPE( loCurSvc ) # "O"
        *** No more services found
        EXIT
      ENDIF
```

If we have an object, we can retrieve the properties and insert them into the appropriate cursor. We can then proceed to retrieve the port information for the current service by calling the form's custom *GetPorts()* method, passing the current service object and the primary key of the record in the cursor.

```
      *** Increment the Service PK Counter
      lnSvcPK = lnSvcPK + 1
      *** Retrieve the properties
      STORE "" TO lcSvc, lcDoc
      WITH loCurSvc
        lcSvc = .Name
        lcDoc = .Documentation
      ENDWITH
      *** Add a record to the Services Cursor
```

```
      INSERT INTO curSvc VALUES ( ;
                  lnSvcPK, ;
                  lcSvc, ;
                  IIF( EMPTY( lcDoc ), "", lcDoc ) )
      *** And now we need to get the Ports for this Service
      ThisForm.GetPorts( loCurSvc, lnSvcPK )
    ENDDO
    RETURN
ENDWITH
```

The *GetPorts()* method is essentially the same as the *GetServices()*, except that it retrieves a different set of properties, and inserts the values into a different cursor.

```
LPARAMETERS toSvc, tnSvcPK
LOCAL loPortObj, loCurPort, lnPortPK, lcName, lcAddr, lcBind, lcTspt, lcDocs
WITH ThisForm
  *** The reader expects these objects to be initialized to 0
  STORE 0 TO loPortObj, lnPortPK
  *** Now find the port(s) associated with the passed in Service
  toSvc.GetSoapPorts( @loPortObj )
  *** Iterate through the Ports object collection object
  DO WHILE .T.
    STORE 0 TO loCurPort
    loPortObj.Next( 1, @loCurPort, 0 )
    IF VARTYPE( loCurPort ) # "O"
      *** No more ports
      EXIT
    ENDIF
    *** Increment the Port PK Counter
    lnPortPK = lnPortPK + 1
    *** Retrieve the properties
    STORE "" TO lcName, lcAddr, lcBind, lcTspt, lcDocs
    WITH loCurPort
      lcName = .Name
      lcAddr = .Address
      lcBind = .BindStyle
      lcTspt = .Transport
      lcDocs = .Documentation
    ENDWITH
    INSERT INTO curPort VALUES ( ;
                lnPortPK, ;
                tnSvcPK, ;
                lcName, ;
                lcAddr, ;
                lcBind, ;
                lcTspt, ;
                lcDocs )
    *** And now we need to get the Methods available on this Port
    ThisForm.GetMethods( loCurPort, lnPortPK )
  ENDDO
ENDWITH
```

As we process each port, we carry on drilling down by calling the form's custom *GetMethods()* and *GetParams()* methods, each of which operates in precisely the same way.

The full list of properties for each level of the WSDL object model is shown in **Table 3**, and while we do retrieve all of them into the cursors (unlike the FoxPro foundation classes), we are only showing a subset in the form.

***Table 3**. WSDL Document Object Model.*

Object	Properties		
Service	• Name	<string>	Service name (= '*Class Library*')
	• Documentation	<string>	Descriptive text for the service
Port	• Name	<string>	Port name (= '*Class Name*')
	• Documentation	<string>	Descriptive text for the port
	• Address	<string>	Location of the port
	• BindStyle	<string>	Binding style (either "*rpc*" or "*document*")
	• Transport	<string>	Protocol used to encode/decode the SOAP message
Operation	• Name	<string>	Operation name (= '*Method name*')
	• Documentation	<string>	Descriptive text for the operation
	• HasHeader	<logical>	Flag indicating presence of a header
	• PreferredEncoding	<string>	Format (e.g. "UTF-8")
	• SoapAction	<string>	Defines the operation required
	• Style	<string>	Operation style attribute (either "*rpc*" or "*document*")
Parts	• CallIndex	< integer>	Index (WSML information)
	• ComValue	<variant>	Defines application data type to be returned
	• ElementName	<string>	Part name in the serializer object
	• ElementType	<string>	Type attribute for the part
	• Encoding	<string>	URI of the encoding specification
	• IsInput	<integer>	Defines whether the part is included in the request only (Parameter = 0), the response only (Return Value = -1) or Both (=1)
	• MessageName	<string>	Message element that contains the mapper object
	• ParameterOrder	<integer>	Part Sequence number (-1 = Return Value)
	• PartName	<string>	Part name in the mapper object
	• VariantType	<integer>	Defines the COM data type for ComValue
	• XMLNameSpace	<string>	URI of namespace associated with the data type

Conclusion

The main objective of this chapter was to demonstrate some of the ways in which you can access information that is available on the Internet from within a Visual FoxPro application. The examples shown here are very simple, but the techniques used are applicable in any context. Although we have only scratched the surface of what is possible, you can see just how easy it is to use Visual FoxPro in this environment.

Chapter 6
Creating Charts and Graphs

It has been said that a picture is worth a thousand words. This is especially true when analyzing trends in financial applications. Viewing the data in a graphical format is usually more meaningful than merely reviewing a bunch of numbers in a spreadsheet. In this chapter, we will explore several mechanisms by which we can generate graphs to be displayed on forms or printed in reports. (Note: Creating graphs for display in Web pages can be accomplished using the Office Web Components. This is covered in Chapter 16, "VFP on the Web".)

Graphing terminology

When we first began working with graphs, we were quite confused by all the terms used to refer to the components of the chart object. The worst thing was that we were unable to find any definition for these terms in any of the documentation. Take, for example, the following excerpt from the MSGraph Help file entry on the series object:

> *Using the Series Object*
>
> *Use **SeriesCollection**(index), where index is the series' index number or name, to return a single **Series** object. The following example sets the color of the interior for series one in the chart.*
>
> *myChart.SeriesCollection(1).Interior.Color = RGB(255, 0, 0)*

Clearly, this is less than helpful if you don't know what a series is. So let's begin with defining a few basic terms. A chart series is a single set of data on the graph. For example, if we create a chart from the data shown in **Figure 1**, each column in the table, excluding the first, would create one series object in the chart's series collection. At least, this is the way it works most of the time. It depends on whether or not the graphing engine plots the series in rows or columns. The default for the MSChart control is to plot the series in columns. However, the default for MSGraph is to plot the series in rows! Since MSGraph is merely a cut-down version of Excel's graphing engine, one would expect Excel to plot the series in rows as well. Surprisingly enough, the default for Excel is "best fit" and it plots the series from whichever are fewer. So, If the data has more rows than columns, Excel uses the data from the columns to create the series objects. Fortunately, the way in which the series are plotted is configurable and can be controlled programmatically.

The way in which a series is represented depends upon the chart type. **Figure 2** shows the four series that are created by the previous sample data when the series are plotted in columns. The data in each series object is represented in this chart type by columns of different colors.

On the other hand, **Figure 3** shows what the same chart looks like when the series are plotted from the data in the rows.

Graphs.ppt - Datasheet

		A	B	C	D	E
		1st Qtr	2nd Qtr	3rd Qtr	4th Qtr	
1	East	20.4	27.4	90	20.4	
2	West	30.6	38.6	34.6	31.6	
3	North	45.9	46.9	45	43.9	
4						
5						

***Figure 1**. Data used to generate a graph.*

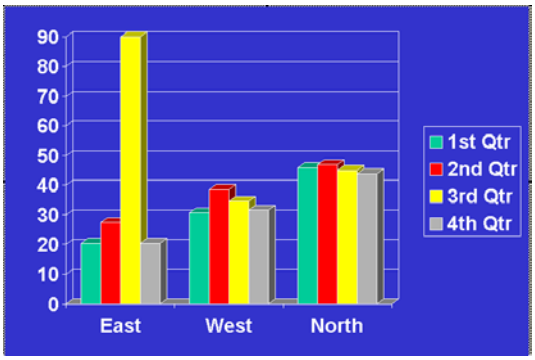


***Figure 2**. 3-D clustered column graph containing four series objects (series in columns).*

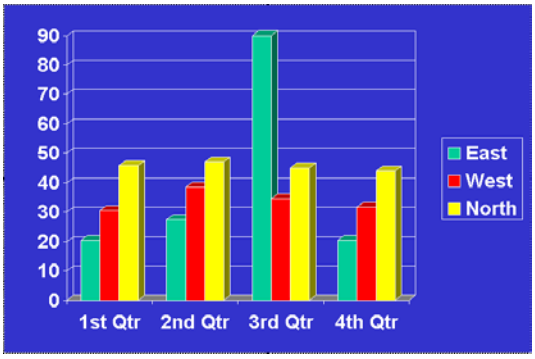


***Figure 3**. 3-D clustered column graph containing three series objects (series in rows).*

Most chart objects contain an axis collection. Two-dimensional charts have an x-axis (horizontal) and a y-axis (vertical). Three-dimensional charts add a z-axis (for depth). These axes are also referred to as:

- *Category axis* When the series data is plotted in columns, this identifies each row in the data that is used to generate the chart. This is usually, but not necessarily, synonymous with the x-axis. In Figure 2, the category

axis displays the names of the regions. In Figure 3, where the series data is plotted in rows, the category axis displays the quarters.

- *Value axis* This identifies the range of values that will be displayed in the chart. This is usually, but not necessarily, synonymous with the y-axis. When defining the scale it is important to ensure that the maximum and minimum values of any series are encompassed by the value axis. In Figure 2 the values range between 0 and 90.
- *Series axis* In three-dimensional charts each series is allocated its own set of spatial coordinates. This is usually, but not necessarily, synonymous with the z-axis. When the series data is plotted in columns, the labels along the series axis correspond to the column headings in the original data. When the series data is plotted in rows, this corresponds to the contents of the first column in the data used to generate the graph. The series axis labels and the legend entries display the same text.

Axes have grid lines and tick lines. The grid lines for the value axis in Figure 2 are the horizontal lines at each interval of 10. The lines that separate the labels on the category axis are the tick lines. These labels are also known as tick labels.

How do I create a graph using MSChart? (*Example: MsChartDemo.scx and CH06.vcx::acxChart*)

The MSChart control is a good starting point for working with graphs because it is a visual control. You can drop it on a form and see how changing its properties affect the appearance of the graph (see **Figure 4**). Unfortunately, Microsoft stopped supporting this control on July 1, 1999 because, being single-threaded, it is incompatible with versions of Microsoft Internet Explorer later than Version 4.0. However, it still ships with Visual FoxPro and, if all you want to do is display a simple graph in a Visual FoxPro form, it is still a good solution.

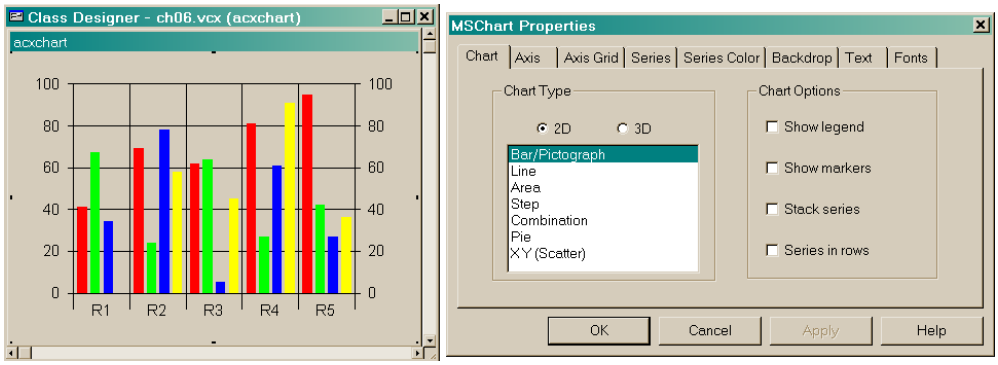


***Figure 4**. MSChart control properties.*

The MSChart control is associated with a *DataGrid* that is used to create the necessary series objects. So in order to get MSChart to display a graph, we first have to populate the

DataGrid, ensuring that we get things in the correct locations. Since the Chart control is a data bound control, we could create an ADO Recordset that contains the data to display, and use it as the Chart control's *DataSource*. When we use a recordset (and only when we use a recordset), the first field is assumed to be the label for the category axis when it holds character data. Otherwise, the first column is treated no differently than any other and is used to define a series object. So, if the labels for your category axis represent numeric values, you must format them as character strings when using an ADO Recordset with the Chart control.

Unfortunately, we cannot bind the Chart control directly to a Visual FoxPro cursor. So, unless we want to create an ADO Recordset from the cursor, we must iterate through the records in our cursor and populate the control's *DataGrid* directly like this:

```
LOCAL lnCol, lnRow
WITH THISFORM.oChart
  *** Set the number of fields in the cursor
  .ColumnCount = FCOUNT( 'csrResults' ) - 1

  *** Set the number of rows to the number of records in the cursor
  .RowCount = RECCOUNT( 'csrResults' )

  *** Populate the DataGrid Object.
  SELECT csrResults
  SCAN
    .Row = RECNO( 'csrResults' )
    FOR lnCol = 1 TO .ColumnCount
      .Column = lnCol
      *** Since the first column is used for the category labels
      *** We must increment our counter
      .Data = EVALUATE( FIELD( lnCol + 1 ) )
    ENDFOR
  ENDSCAN
ENDWITH
```

Note that when we manually populate the Chart's *DataGrid* like this, the labels for the category axis do not automatically come from the first column of the data. In fact, used this way, the *DataGrid* can only contain the actual values that will be used to generate series objects. Labels are added by explicitly setting the properties for them on both the category and value axes.

Having populated the grid, we can set the properties that control the output. There are an awful lot of these, but the most important ones for explaining what a graph shows are:

- *RowLabel*: The labels collection for the category axis.
- *ColumnLabel*: The labels collection for the value axis.
- *AxisTitle.Text*: The title for the axis title object.
- *AxisTitle.VtFont.Size*: The font size for the axis title object.

The sample form creates three different graphs on the fly and displays them using the MSChart control. To make the graph generation process extensible and maintainable, we store the graph definitions in a free table called QUERIES.DBF with the structure shown in **Table 1**.

***Table 1**. Structure of metadata used to store graph definitions.*

Field name	Type	Length	Description
cQueryName	C	20	Keyword used to look up the record in QUERIES.DBF.
cQueryDesc	C	50	Query description for use in end user displays.
cPopupForm	C	80	Name of pop-up form used to obtain values for the query's ad hoc where clause if one is specified.
nChartType	I		Type of chart to generate (e.g. 3-D Bar, 2-D Line) defined by one of the chart type constants in MSCHRT20.H.
nGraphType	I		Serves the same purpose as nChartType when used to generate graphs using MsGraph (we need both because the constants have different meanings in MSGraph and MSChart).
mQuery	M		The query used to obtain the data that will be used to generate the chart. This may include expressions like "`WHERE Customer.cust_id = '<<pcParm1>>'`" to specify an ad hoc `WHERE` clause because the `TEXTMERGE()` function will be used before the query is run.
cMethod	C	80	The name of a form method to run after the query is run to massage the result cursor.
cTitleX	C	50	Title for the x-axis.
cTitleY	C	50	Title for the y-axis.
cTitleZ	C	50	Title for the z-axis.

The form has seven custom methods that use the data in QUERIES.DBF to gather any required parameters from the user and generate the graph (see **Table 2**).

***Table 2**. MSChartDemo.scx custom methods.*

Method name	Description
MakeGraph	Called by the *onClick()* method of the "Create Graph" command button, this is the control method that generates the graph.
DoQuery	Called by the form's *MakeGraph()* method, uses the passed parameter object to run the query contained in the mQuery field of the current record in QUERIES.DBF. It always places the query results in a cursor named *csrResults* so that we can write generic code in the form to handle the results of different queries.
GetQueryParms	Called by the form's *MakeGraph()* method. This method instantiates the form specified in the cPopUpForm field of the current record in QUERIES.DBF. The pop-up form returns a parameter object, which is passed back to the *MakeGraph()* method.
MakeMonthColumns	Called by *MakeGraph()* when it is specified in the cMethod field of QUERIES.DBF. It takes the contents of *csrResults* , which is a "vertical" structure with a single record for each month, and converts it into a "horizontal" structure with one field for each month in a single record.
MakeYearColumns	Called by *MakeGraph()* if it is specified in the cMethod field of QUERIES.DBF. It takes the contents of *csrResults*, which is a "vertical" structure with a single record for each year, and converts it into a "horizontal" structure with one field for each year in a single record. The number of fields in the new structure depends upon the range of distinct years contained in the original cursor.
PopulateDataGrid	Called by *MakeGraph()* to populate the graph's *DataGrid* object from the information in *csrResults*.
SetAxisTitles	Uses the information in the title fields in QUERIES.DBF to set the axis titles. Also sets the fonts for the titles and labels.

As you can see from Table 2, the form's custom *MakeGraph()* method controls the whole process. It passes any required parameters to the form's custom *DoQuery()* method. *DoQuery()*, as its name implies, runs the query from QUERIES.DBF to generate the cursor *csrResults* that holds the data used to generate the graph. Next, when a method name is specified in *Queries.cMethod*, *MakeGraph()* first checks to make sure that the method exists and then calls it. The *csrResults* cursor, generated by the *DoQuery()* method, is used by *PopulateDataGrid()* to pass data to the chart like this:

```
LOCAL lnCol, lnRow
WITH THISFORM.oChart

  *** Set the chart type
  .ChartType = Thisform.cboChartType.Value

  *** Set the number of fields in the cursor
  .ColumnCount = FCOUNT( 'csrResults' ) - 1

  *** Set the number of rows to the number of records in the cursor
  .RowCount = RECCOUNT( 'csrResults' )

  *** Populate the DataGrid Object.
  SELECT csrResults
  SCAN
    .Row = RECNO( 'csrResults' )

    *** Set up the label for the category axis
    .RowLabel = EVALUATE( FIELD[ 1 ] )

    *** Populate the data grid with the numeric data
    FOR lnCol = 1 TO .ColumnCount
      .Column = lnCol
      .Data = EVALUATE( FIELD( lnCol + 1 ) )
    ENDFOR
  ENDSCAN
  FOR lnCol = 1 TO .ColumnCount
    .Row = 1
    .Column = lnCol
    .ColumnLabel = ALLTRIM( FIELD( lnCol + 1 ) )
  ENDFOR
ENDWITH
```

The act of populating the *DataGrid* forces the chart to display on the form; however, at this point all it has is the raw data and axis labels. The final steps in the *MakeGraph()* process are to call *SetAxisTitles()* and then tidy up the display by setting the chart's *Projection*, *Stacking*, and *BarGap* properties to values that are more suitable than the defaults that MSChart supplies when the chart is re-drawn.

Note that the *PopulateDataGrid()* method manipulates the *Data* property of the chart object directly. However, if you open MSCHRT20.OCX in the object browser, you will not be able to find this property. Apparently it is not exposed by the type library. However, it is documented in the Help file (MSCHRT98.CHM) and certainly appears to be present and available. It is used to get, or set, a value at the current data point in the data grid of a chart. A data point is made current by setting the chart's *Row* and *Column* properties to its coordinates. By the

way, these properties do not appear in the object browser either, even though they too are listed in the Help file.

As you can see, getting a graph into a form is actually pretty easy with MSChart. However, including an MSChart graph in a printed report is not. MSChart does have an *EditCopy()* method that copies the chart object to the clipboard in metafile format, but there is no easy way to transfer the metafile from clipboard to disk without using additional software. Nor can you insert the chart object into a *General* field, so it cannot be included in a printed report that way either. So if printing is a requirement, you need to use something other than MSChart.

How do I create a graph using MSGraph? (*Example: MsGraphDemo.scx*)

MsGraph is actually a cut-down version of the Microsoft Excel graphing engine. You can see this if you open GRAPH9.OLB in the object browser and expand the *enums* node (see **Figure 5**).

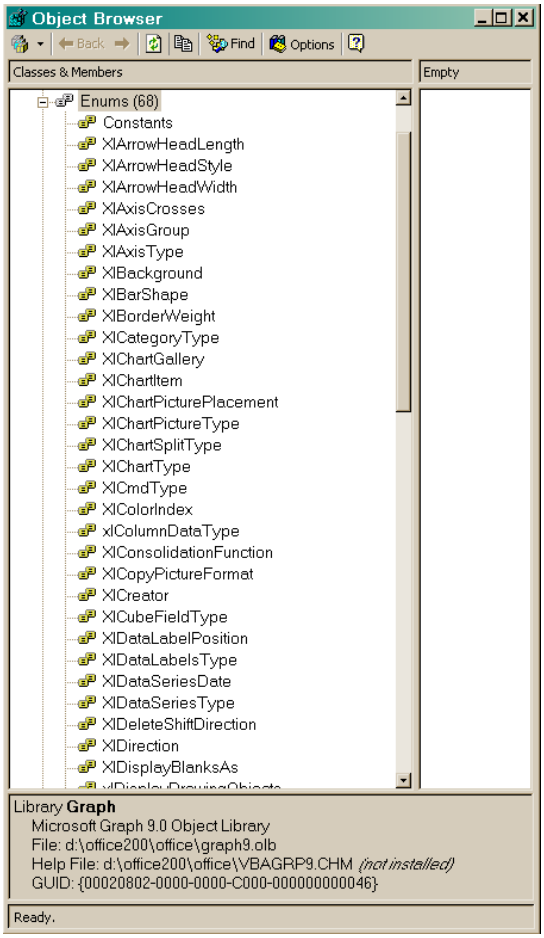


***Figure 5**. MSGraph enums in the object browser.*

Notice that all the constant names begin with "Xl". There is a very good reason for this—they are the same names and values that are used by Excel's graphing engine. So, if you start out using MSGraph to create your graphs and later decide to move to Excel Automation, you should find that most of the code to manipulate the graph will run with no modification.

Having a much simpler object model than Excel (see **Figure 6**), MSGraph is lighter and quicker to instantiate and therefore the results appear more quickly too.

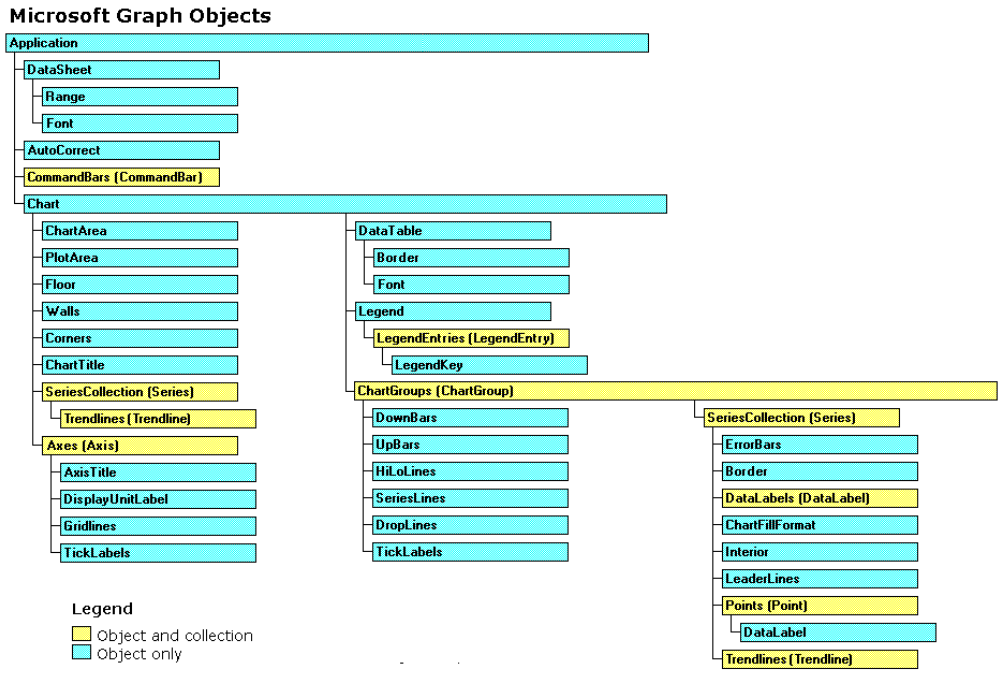


***Figure 6**. MSGraph object model.*

MSGraph gives you much more control over the graph's appearance than MSChart does. It is also possible to include graphs in printed reports if you use MSGraph to generate them because they *can* be added to, and printed directly from, a *General* field in a table or cursor.

One unpleasant surprise that we encountered when working with MSChart and MSGraph was the lack of consistency between the two. Not only did the properties, methods, and events have different names, even the constants had different meanings! For example, a chart type of "1" in MSChart produces a 2-dimensional bar graph. In MSGraph, this is an area graph.

The easiest way that we have found to manipulate MSGraph is to store a "template" graph in a *General* field of a table. Then, whenever we need to create a particular graph, we drop an OleBound Control on a form and set its *ControlSource* to the *General* field in a cursor created from that table. This has several benefits:

- It avoids contention when several users try to create a graph at the same time, since each user has his own cursor.
- It does not require multiple graphs to be stored. After all, graphs are generally used to depict current trends and, as such, are usually generated "on the fly," so it makes no sense to store them permanently in a table or on disk.
- Using an OleBound control gives us direct access to the graph through its properties. This means we can manipulate the graph programmatically while the form is invisible and then display or print the final result.

The sample form uses this technique. Notice that we are using the same data-driven methodology that we used with the MSChart control. Thus, the custom *MakeGraph()* method in the sample form controls the graph generation process and calls the following supporting methods:

- *GetQueryParms* — Pulls up the pop-up form to gather any values required for the query's ad hoc where clause if a pop-up form is specified in the cPopupForm field of QUERIES.DBF.
- *DoQuery* — Creates a cursor named *csrResults* by running the query specified in QUERIES.DBF.
- *UpdateGraphData* — Constructs the proper format string to use with **APPEND GENERAL DATA** and issues the command.
- *FormatGraph* — Sets various graph properties such as axis titles, tick label fonts, and so on.

Two new custom methods, *UpdateGraphData()* and *FormatGraph()*, use the cursor to render the appropriate graph. The only differences from the MSChart sample are in the details of the code that is used to generate and format the graph object.

The form's custom *UpdateGraphData()* method uses the native Visual FoxPro **APPEND GENERAL** command with the **DATA** clause to update the graph in the *General* field of a cursor named *csrGraph*. This cursor is created from the table VFPGRAPH.DBF in the form's *Load()* method. (Remember, the VFPGRAPH.DBF table has only one record, which stores the template graph and which is never updated.) In order to use this form of **APPEND GENERAL**, the data must be in "standard clipboard" format, which means that the fields are separated by tab characters, and the records are separated by carriage returns. The first part of the method deals with converting the data from *csrResults* into the correct format.

```
LOCAL lcGraphData, lnFld, lnFieldCount

*** Make the oleBoundControl invisible
*** and unbind it so we can update the general field
Thisform.LockScreen = .T.
Thisform.oGraph.ControlSource = ''

*** Now build the string we need to update the graph
*** in the general field
```

```
lcGraphData = ""
SELECT csrResults
lnFieldCount = FCOUNT()

*** Build tab-delimited string of field names:
FOR lnFld = 1 TO lnFieldCount
  lcGraphData = lcGraphData + FIELD( lnFld ) ;
    + IIF( lnFld < lnFieldCount, CHR( 9 ), CHR( 13 ) + CHR( 10 ) )
ENDFOR

*** Concatenate the data, converting numeric fields to character:
SCAN
  FOR lnFld = 1 TO lnFieldCount
    lcGraphData = lcGraphData + TRANSFORM( EVALUATE( FIELD( lnFld ) ) ) + ;
      + IIF( lnFld < lnFieldCount, CHR( 9 ), CHR( 13 )  + CHR( 10 ) )
  ENDFOR
ENDSCAN

GO TOP IN csrResults

*** OK, ready to update the graph
SELECT csrGraph
APPEND GENERAL oleGraph CLASS "MsGraph.Chart" DATA lcGraphData
```

Having updated the *General* field, we can bind the control on the form directly to the cursor and set the following properties:

- *ChartType*: Determines the type of graph. Values are defined in GRAPH9.H.
- *Application.PlotBy*: Determines whether series objects are generated from rows, or columns in the data.

```
WITH Thisform.oGraph
  *** Reset the controlSource of the OleBound control
  .ControlSource = "csrGRaph.oleGraph"

  *** Set the chart type
  .object.ChartType = Thisform.cboGraphType.Value

  *** Set the data to graph the columns as the series
  *** Unless, of course, this is a pie chart
  IF NOT INLIST( .ChartType, xl3DPie, xlPie, xlPieOfPie, xlPieExploded, ;
      xl3DPieExploded, xlBarOfPie )
    .Object.Application.PlotBy = xlColumns
  ELSE
    .Object.Application.PlotBy = xlRows
  ENDIF
ENDWITH

Thisform.LockScreen = .F.
```

It is evident from this code listing that we ran into a couple of problems when creating the sample form. First, we discovered that the **APPEND GENERAL** command refused to update the graph in the general field while it was bound to the OleBound control. We finally had to unbind the control before issuing the command and re-bind it afterward. Second, we had to explicitly tell MSGraph to use the columns as the data series for all charts except pie charts,

overriding the default behavior, which is data series in rows and which can produce some very odd-looking graphs (especially when you have only one row of data!).

That is all that must be done to generate and display the graph. However, we found that the default values produced ugly graphs and so we needed to set some additional properties to improve the appearance. The properties and methods available for MSGraph are, to say the least, comprehensive. The full list is included in VBAGRP9.CHM, the Help file for MSGraph. However, the actual documentation is rather sparse, so, once you are certain that the item you are interested in actually exists in the MSGraph object model, we suggest that you look it up in the Excel documentation—which is slightly better. Remember, MSGraph is just a cut-down version of Excel's graphing engine.

While it is beyond the scope of this chapter to show you how to manipulate all of these properties, the custom *FormatGraph()* method shows how manipulating a few of the graph's properties changes its appearance.

The first thing that we want to do is to set the axis titles and fonts. However, not all chart types have an axes collection (most notably Pie charts). The chart object exposes a *HasAxis()* method that, despite being referred to in the documentation as a *Property*, accepts a constant that identifies the axis type and returns a logical value. You would be forgiven for thinking that we could use this to tell us whether a given axis exists. However, it turns out that if the graph does not have an *Axes* collection, trying to access this "property" simply causes an OLE error. So we have no alternative but to check the graph type explicitly:

```
IF NOT INLIST( .ChartType, xl3DPie, xlPie, xlPieOfPie, ;
               xlPieExploded, xl3DPieExploded, xlBarOfPie )
```

and only if the chart has axes do we then proceed to configure them, by setting the following properties for each axis object in the *Axes* collection:

- TickLabels.Font.Size
- HasTitle
- AxistTitle.Text
- AxisTitle.Font.Size

```
WITH .Axes( xlCategory )
  .TickLabels.Font.Size = 8
  lcTitleText = ALLTRIM( Queries.cTitleX )
  IF NOT EMPTY( lcTitleText )
    .HasTitle = .T.
    .AxisTitle.Text = lcTitleText
    .AxisTitle.Font.Size = 10
  ELSE
    .HasTitle = .F.
  ENDIF
ENDWITH
```

For the chart types that don't have an *Axes* collection, we only need to set the following properties on the chart object:

- *HasTitle*
- *ChartTitle.Text*
- *ChartTitle.Font.Size*

But we also need to call an additional method, *ApplyDataLabels()*, to assign labels to the pie chart segments.

One word of caution. Even though a given property or method is defined as being part of the object model, not all properties and methods are always available. As we found with the *HasAxis()* method, it is imperative to ensure that a specific instance of the graph actually has the required property or method before trying to access it. If, for any reason, it is not available, MSGraph hands you back a nasty OLE error.

How do I create a graph using Excel Automation? (*Example: ExcelAutomation.scx*)

Producing graphs represents only a tiny fraction of what you can do when you harness the power of Excel in your Visual FoxPro applications. While VFP is an excellent tool for manipulating data, it is definitely not the best tool when it comes to dealing with complex mathematical formulae. An entire book could be devoted to the topic of Excel Automation (in fact, several have), and a complete discussion of its capabilities is beyond the scope of this chapter. For specific examples using Visual FoxPro, see *Microsoft Office Automation with Visual FoxPro* by Tamar E. Granor and Della Martin (Hentzenwerke Publishing, 2000, ISBN: 0-9655093-0-3).

The example form (see **Figure 7**) uses the same data-driven methodology that we have used with MSGraph and MSChart. The difference is that instead of formatting our data and feeding it directly to the graphing tool, we now have to feed the data to Excel and then instruct it to create a graph using that data. To do this we added a custom *AutomateExcel()* method that first creates an instance of Excel, then opens a workbook and populates a range in the active worksheet with the data from our results cursor:

```
*** create an instance of excel.
loXl = CREATEOBJECT( 'Excel.Application' )

*** Now add a workbook so we can populate the active worksheet
*** with the data from the query results
loWB = loXl.Workbooks.Add()

*** Now fill in the data
WITH loWb.ActiveSheet

  *** Give it a name so we can reference it
  *** after we add a new sheet for the chart
  .Name = "ChartData"

  *** Make sure we have the field names in the first row of the work sheet
  *** we do not want the field name for the first column which is used to
  *** identify the categories
  FOR lnCol = 2 TO FCOUNT( 'csrResults' )
```

```
   *** Convert the field number into an Excel cell designation
   *** We can do this easily because 'A' has an ascii value of 65
   lcCell = CHR( lnCol + 64 ) + "1"

   *** Go ahead and set its value
   .Range( lcCell ).Value = ALLTRIM( FIELD( lnCol, 'csrResults' ) )
ENDFOR
```

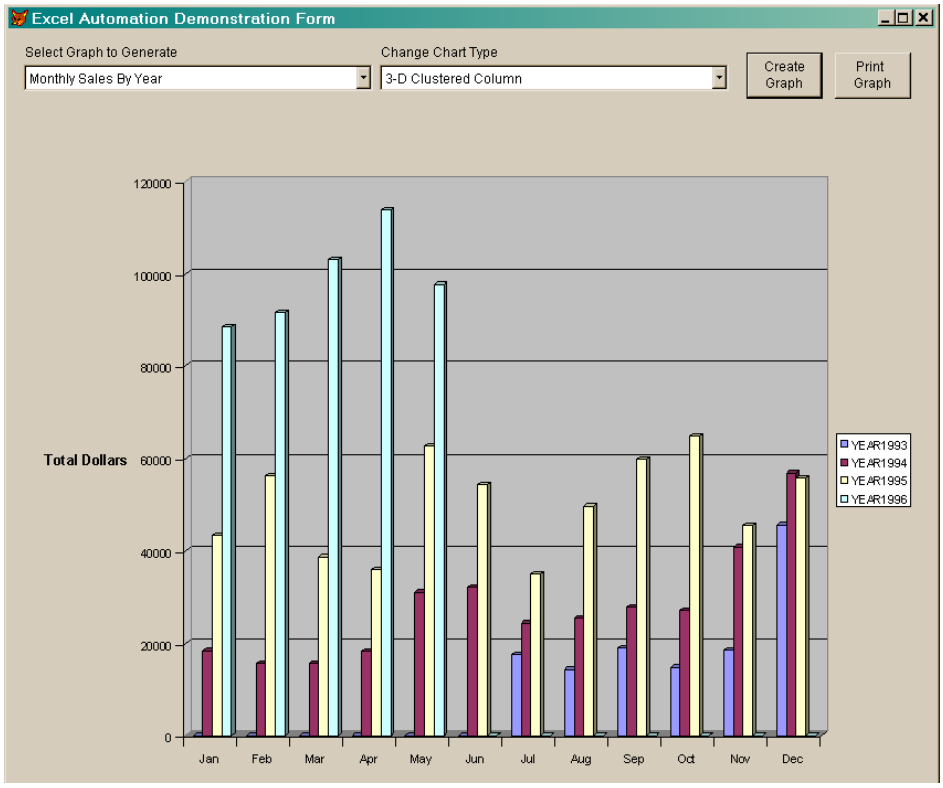


***Figure 7**. Excel Automation sample form.*

Populating the cells in the worksheet is a little trickier than populating the *DataGrid* of the MSChart control, or sending data to MSGraph, because the cells in an Excel spreadsheet are identified by an alphanumeric key. The columns are identified by the letters A through Z while the rows are identified by their row numbers. So, to access a particular cell, you must use a combination of the two. For example, to identify the cell in the first row of the first column of the spreadsheet, you identify it as Range('A1'). As you can see in the following code, it is easy enough to convert a column number to a letter because the letter "A" has an ASCII value of 65. So all we need to do is add 64 to the field number in our cursor and apply the **`CHR()`** function to the result.

```
  *** Now just scan the cursor and populate the rest of the cells
  SELECT csrResults
  SCAN
    FOR lnCol = 1 TO FCOUNT( 'csrResults' )

      *** Get the cell in the worksheet that we need
      *** Since the first row has the column headings, we must
      *** start in the second row of the worksheet
      lcCell = CHR( lnCol + 64 ) + TRANSFORM( RECNO( 'csrResults' ) + 1 )

      *** Go ahead and set its value
      .Range( lcCell ).Value = EVALUATE( FIELD( lnCol, 'csrResults' ) )
    ENDFOR
  ENDSCAN
  GO TOP IN csrResults
ENDWITH
```

This code works, but there is one major problem with it. It is slow! Poking values into individual cells in a spreadsheet inside of a tight loop is not the most efficient way to get the job done. Fortunately, we can use the *DataToClip()* method of the Visual FoxPro application object to copy the data in our cursor to the clipboard. Then we can use the active worksheet's *Paste()* method to insert the data from the clipboard into the spreadsheet. Using the following modified code yields an improvement in performance of up to 60% depending on the volume of data being sent to Excel. Another benefit of using our modified version of the code to send data to Excel is that there is less of it. Less code means fewer bugs.

```
WITH loWb.ActiveSheet
  *** Give it a name so we can reference it
  *** after we add a new sheet for the chart
  .Name = "ChartData"

  *** Get the number of columns
  lnFldCount = FCOUNT( 'csrResults' )

  *** Add one because copying the data to the clipboard
  *** adds a row for the field names
  lnRecCnt = RECCOUNT( 'csrResults' ) + 1
  SELECT csrResults
  GO TOP

  *** Copy to clipboard with fields delimited by tabs
  _VFP.DataToClip( 'csrResults', RECCOUNT( 'csrResults' ), 3 )

  *** Get the range of the data in the worksheet
  lcCell = 'A1:' + CHR( 64 + lnFldCount ) + TRANSFORM( lnRecCnt )

  *** And paste it in
  .Paste( .Range( lcCell ) )

  *** But now we have to make sure that cell A1 is blank
  *** Otherwise the chart is not created correctly
  .Range( "A1" ).Value = ""
  GO TOP IN csrResults
ENDWITH
```

After we transfer the data from the cursor to the spreadsheet, we are ready to create the graph. This is done by adding an empty chart object to the workbook's charts collection and telling it to generate itself. The chart object's *SetSourceData()* method accepts a reference to the range that contains the data together with a numeric constant that specifies how to generate the series objects (that is, from rows or columns). To ensure that the display is in the correct format, the *AutomateExcel()* method forces the chart object's *ChartType* property to the correct value:

```
loChart = loWB.Charts.Add()

*** Set the data to graph the columns as the series
*** Unless, of course, this is a pie chart
IF NOT INLIST( Thisform.cboGraphType.Value, xl3DPie, xlPie, ;
         xlPieOfPie, xlPieExploded, xl3DPieExploded, xlBarOfPie )
  lnPlotBy = xlColumns
ELSE
  lnPlotBy = xlRows
ENDIF

WITH loChart
  *** Generate the chart from the data in the worksheet
  .SetSourceData( loWB.Sheets( "ChartData" ).Range( lcCell ), lnPlotBy )

  *** Set the chart type
  .ChartType = Thisform.cboGraphType.Value
```

At this point we have a chart in an Excel spreadsheet, but what we really want is to display it in a Visual FoxPro form. The easiest way to do this is to use the chart's *SaveAs()* method to save it to a temporary file. We can then use the **APPEND GENERAL** command to suck the chart into the *General* field of the cursor that was created in the *Load()* method of the demonstration form. Once the graph is safely in the *General* field, we can quit Excel, erase the temporary file, and bind the OleBound control on the form to the cursor's *General* field. The last part of the *AutomateExcel()* method does exactly that:

```
  *** Save to a temporary file
  lcFileName = SYS( 2015 ) + '.xls'
  loChart.SaveAs( FULLPATH( CURDIR() ) + lcFileName )
ENDWITH

*** and quit the application
loXl.Quit()

*** insert the graph into the general field in the cursor
SELECT csrGraph
APPEND GENERAL oleGraph FROM ( lcFileName ) CLASS "Excel.Chart"

*** and clean up
ERASE ( lcFileName )

WITH Thisform
  *** Reset the controlSource of the OleBound control
  .oGraph.ControlSource = "csrGraph.oleGraph"
  .LockScreen = .F.
ENDWITH
```

Now that the graph is bound to the OleBound control on the form, we can manipulate its appearance in much the same way that we did for MSGraph. As a matter of fact, the code in the form's custom *FormatGraph()* method is almost identical to the code in the previous example. Other than changing references to *Thisform.oGraph.Object* to *Thisform.oGraph.Object. ActiveChart*, all we had to change for our Excel Automation sample was to add this code:

```
*** Now set the axes at right angles for 3-d bar, column, and line charts
IF INLIST( .ChartType, xl3DColumnClustered, xl3DColumnStacked, ;
           xl3DColumnStacked100, xl3DBarClustered, xl3DBarStacked, ;
           xl3DBarStacked100, xl3DLine )
  Thisform.oGraph.Object.ActiveChart.RightAngleAxes = .T.
ENDIF
```

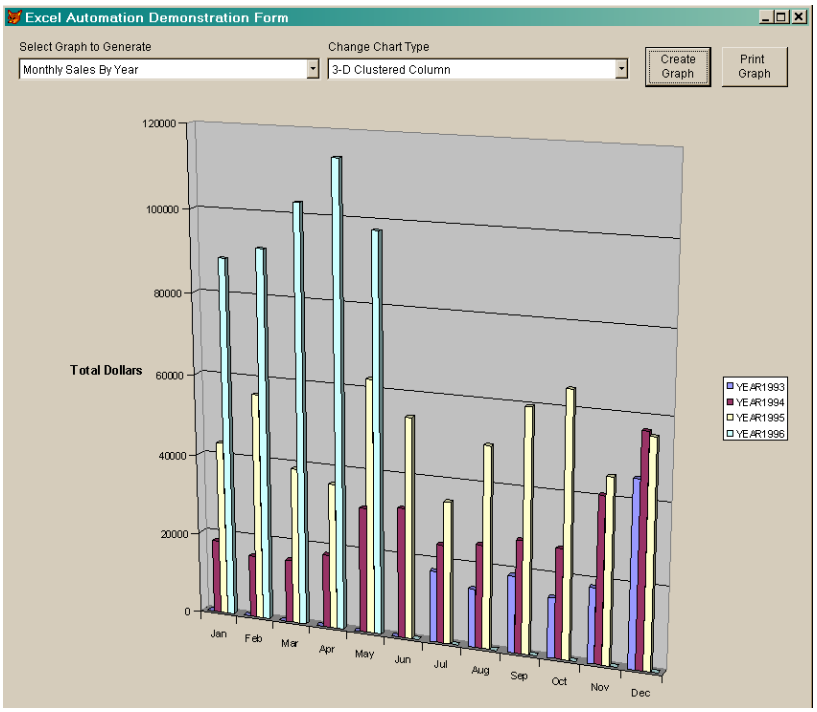


***Figure 8**. Default perspective of 3-D graph generated by Excel.*

This is because MSGraph, by default, creates a pretty 3-D graph with the axes at right angles to each other. Excel does not. Before we added this code, the graph looked like **Figure 8**. As you can see, it had an unappealing ragged appearance.

The graph object has a huge numbers of properties and methods that you can use to manipulate its appearance, which are described in the Excel 2000 Help file, VBAXL9.CHM. In addition to the documentation, our sample form makes it easy for you to experiment with the effect of changing properties. All you have to do is run the form, click the "Create Graph" button, and type this in the command window:

```
o = _Screen.ActiveForm.oGraph.Object.ActiveChart
```

You can then call the methods of the chart object or change its properties and immediately see either an OLE error or a change in the appearance of the graph in the form.

Using Visual FoxPro 7 makes the discovery process even easier because of IntelliSense. Once you have a reference to the chart object, you can see a list of all the methods and properties that apply (see **Figure 9**).

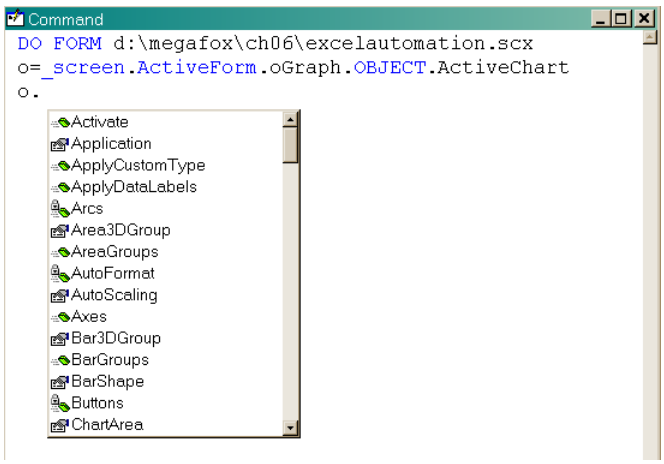


***Figure 9**. Exploring the Excel object model from the VFP 7.0 command window.*

Conclusion

A picture is indeed worth a thousand words. Including graphs and charts in your applications, whether displayed in a form or printed in a report, adds a lot of pizzazz and a professional look and feel. In the past, adding this functionality was more than a little painful because of the lack of documentation and good working examples. We hope that this chapter has given you a better starting point than we had when we wrote it, and that you can use the examples to create some really spectacular graphs of your own.

Chapter 7
New and Improved Reporting

Application Reporting is a fundamental aspect of database development. While users could just have data collection repositories that accumulate an abundance of information, practically speaking, the information is nearly useless unless the users have some way to look at it, massage it, and have it presented to them. There are many ways of doing that with the user interface. The primary way is through reporting tools like the Visual FoxPro Report Designer and Crystal Reports.

Reporting is one way our customers analyze their information. It is important to provide easy-to-read and informative reports. Sometimes this is difficult within the limitations of the reporting tools. There is a science to matching the capabilities of the tools we have today with the high expectations our customers have with their reporting needs. It is these challenges we seem to face often in development. This chapter aims to present solutions to a few specific challenges with respect to reporting.

This chapter is split into two major sections. The first quarter of the chapter will deal with the native Visual FoxPro Report Designer. We devoted two chapters in *1001 Things You Wanted to Know About Visual FoxPro*, but we have found a few more tips and tricks up our sleeve. The last three quarters of the chapter will discuss and demonstrate how developers can integrate Crystal Decisions' powerful Crystal Reports with Visual FoxPro and how it addresses a number of limitations we have seen in Visual FoxPro's designer. Crystal Reports has been around for a number of years now and has finally matured into a component technology that can easily be integrated with Visual FoxPro-based applications.

Visual FoxPro Report Designer

What are the new features in the Visual FoxPro 7 Report Designer?

Visual FoxPro developers have voiced displeasure over the years that the Report Designer has not been significantly improved. It is true, and there are a number of tools that have evolved reporting to a more sophisticated level. While the Visual FoxPro 7 Report Designer does not have anything revolutionary, there are some nice enhancements that are worth mentioning.

The big improvement is not in the actual designer, but in the output to the Windows print spooler. In previous versions of Visual FoxPro we saw "Visual FoxPro" as the name of the item being printed. This caused some concern for developers who were not interested in their customers seeing a Microsoft development product printing instead of their custom application. Even more important, all the reports were named the same so there was no way for the customers to know which report was printing and where it was in the queue. Visual FoxPro 7 now adds the report or label file name to the printer queue (see **Figure 1**).

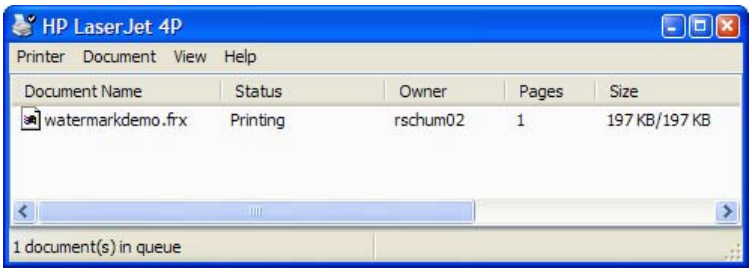


***Figure 1**. The report name is now reflected in the print spooler instead of "Visual FoxPro".*

The Report Designer has been "keyboard enabled." A developer, who is keyboard-centric as opposed to mouse-centric, will find a number of enhancements to the designer. Here is a complete list of changes:

- The new Insert Control item in the Report menu has a submenu with identical controls found on the Report Controls toolbar. Selecting a control from the menu adds the selected report object to the upper left corner of the report. The expression object is added after the expression is filled in. Once the object is added, you can move it around with the arrow keys. Text objects are dropped directly on the report, and you can stop editing the label by pressing the Escape key.
- The Report Designer has improved keyboard navigation. Press Ctrl-Tab to toggle in and out of a new "tab" mode. When you are in tab mode, press Tab to move to the next object and Shift-Tab to move to the previous object. Press Ctrl-E to edit the text of the selected label. One of the problems is that there is no tab order setting in the report, so there can be a random feel depending on how you added objects to the report.
- The new Bands item in the Report menu displays a dialog to select a particular band. The band dialog is displayed so you can edit the band properties.
- The new Background Color and Foreground Color items in the Format menu allow you to set the colors for the selected objects in the report.

We are not sure how practical printing a report with more than 65,000 pages is, but the Report Designer limit is increased from 9,999 to 65,534. So if your users want to be responsible for more of the Brazilian rainforests to be cut down, we now have this functionality.

If you are running a Middle Eastern version of Windows, you might be interested in the new Format menu option Reading Order. It is the reporting equivalent to the *RightToLeft* property set on some controls. It determines if the text is displayed from the right side, moving left instead of the default way of starting at the left and reading toward the right. It is disabled for non-Middle Eastern versions of Windows.

How do I prompt for a printer from preview mode?

The preview mode allows users to see what the report looks like before it is printed. The display of the report is controlled by the default Windows printer. If the user presses the printer button on the Report Preview toolbar, the report is printed on the default printer as well. What if the user wants the report printed on a different printer than the default printer without changing the default printer while the report is being previewed? Keep reading to find the simplest of solutions.

Developers are used to the following syntax when generating a Visual FoxPro report to the previewer:

```
REPORT FORM WaterMarkDemo NOCONSOLE PREVIEW
```

If you want to allow the user to preview the report and to have the option to select a printer, use this syntax:

```
REPORT FORM WaterMarkDemo TO PRINTER PROMPT PREVIEW
```

The syntax reads like a bad contradiction. It is also important to note that the **PREVIEW** clause must follow the **TO PRINTER**. Reversing the syntax will generate error 1306 (Missing Comma (,)), which is not very intuitive. This might be a common problem for developers since the **REPORT FORM** command documentation syntax and the IntelliSense information tip show **PREVIEW** before **TO PRINTER**.

Once the user presses the Print Report icon, the select printer dialog is presented. It is our experience that the report preview mode is blanked out; your mileage might vary depending on the printer and video drivers. The user also has the option to cancel the report by canceling out of the print dialog.

How do I print watermarks on a report? (*Example: **WatermarkDemo.frx/frt***)

A watermark is a light graphic image that is shadowed on the paper (see **Figure 2**). They are a background that is placed on the page. They typically encompass the entire page. Our first inclination is to just add a graphic image and stretch it across all the bands of a report, just like we do with lines and boxes. If you give this a try you will see that it does not work. The graphic stays the same size as you added it in the Report Designer; it does not stretch as you might expect. So how can we add watermarks to a report if stretching a graphic across the bands does not work? Follow along with the steps in this section.

You start the report development by creating the entire report. You will want to add the graphic watermark last. There are a number of issues noted later in this section discussing why you will want to work this way. The watermark is added as a graphic image, which is straightforward. Drop a report Picture/ActiveX control on the report, select the file name, and mark the sizing to “Scale Picture, Retain Shape” or “Scale Picture, Fill the Frame.”

It is important that we add a disclaimer to the rest of this section. You need to make backups of the FRX and FRT files before proceeding. We are about to do something that is not supported by Microsoft. We will be leaving the comforts of the Report Designer and hacking the underlying metadata. You can accidentally (or purposely) disable the metadata when working in the FRX/FRT files. This is the source code to your application. Safeguard it before tinkering with it—hacking can be powerful and at the same time very dangerous.

Now we get to learn a bit about the insides of the FRX metadata table. We are only going to discuss the needed changes to the graphic image used for the watermark. It is outside of the scope of this discussion to detail the other records in the metadata. The first item to reveal is that items in the report metadata are in units of 1/10,000 of an inch. Each item that is sized has height and width properties. These are stored in the Height and Width columns in the FRX. All vertical and horizontal positions are also stored in 1/10,000 of an inch.

```
USE WatermarkDemo.frx EXCLUSIVE
BROWSE LAST
```

Opening the report metadata exclusive is not required, but practical in a team development environment. The Report Designer will not open the report if you are hacking into it, but you don't want other developers hacking the same report as you are making manual changes.

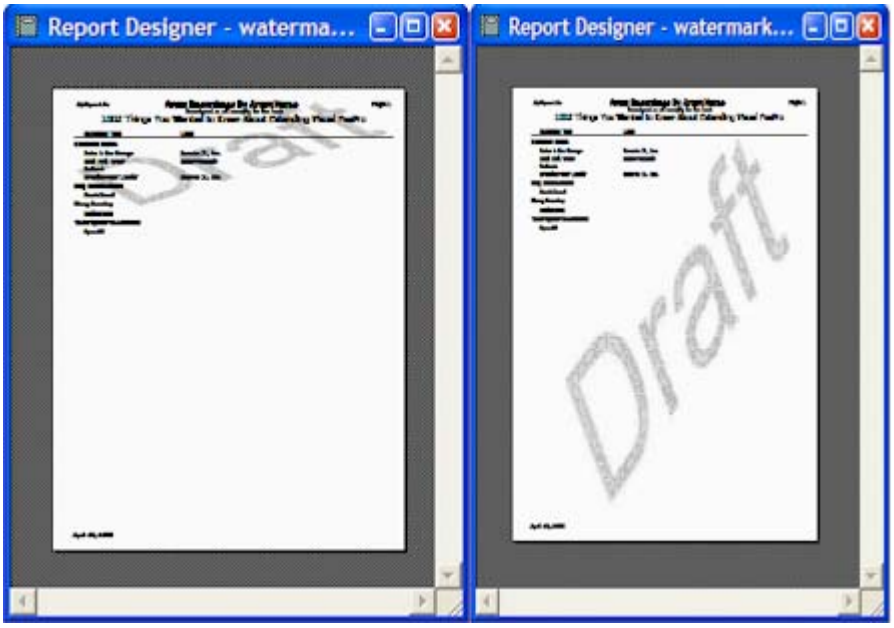


***Figure 2**. The report on the left is the watermark without the "report hack" to make it size right. The report on the right is using the technique to get the watermark to stretch the entire page length.*

You will need to look for the watermark graphic record in the report metadata. It will have a value of 17 in the ObjType column and the Picture column will contain the name of the graphic image. Once you locate the record you will have to look at four columns: VPos (vertical position), HPos (horizontal position), Height, and Width. The vertical and horizontal positioning can be handled in the designer, but you can tweak it here in the table if you need finer or more accurate control. Note the unit of measure discussed earlier in this section. If you want the graphic to have a margin of one inch, you need to put 10000.000 in the VPos and

HPos columns. This positioning is set from the upper left side of the report. You are defining the left and top margins with these settings.

The other important work during the hacking session is to adjust the Height and Width properties of the graphic image. If you want the 8.5 inch by 11 inch paper to be filled with a one inch margin on the paper, you need to calculate the height by taking 11 inches and subtracting two inches (top and bottom). This leaves nine inches times 10,000 units per inch, meaning that the Height column needs a value of 90,000. The same type of calculation is made for the width. Eight and a half inches, minus two for the desired margins is six and a half, times 10,000 gives 65,000. At this point you will need to compensate for the report's left margin if you added one. Take the size of the margin in inches and multiply it by 10,000. Subtract this number from the previous width. If the left margin on the report is 0.2 inches, subtract 2000 from the calculated width. In the example used, we would subtract 2000 from the 65,000.

Overall you might find yourself refining the positions and the width depending on the graphic you are using for the watermark. Each graphic might have its own margins that would throw your calculations off.

There are a couple of issues when working with watermarks. The graphics can get in the way when aligning expression and text objects in the various bands. Lassoing any report objects is likely to select the watermark, so wait to add the watermark until the report is finalized if at all possible. Another reason to wait until the end to add the watermark is that you will be sizing bands and the watermark can get undersized or oversized for the designer and it can get difficult to see or grab the graphic to move it out of the way. Changing the size or any property of the graphic will also require you to reopen the FRX as a table and reset the sizes. One last issue to be noted and something you need to be concerned with if you are developing international applications is that paper sizes are different in various countries. If you are developing the application in Argentina, for instance, on A4 paper and you deploy your applications in the United States with Letter formatted paper, the images could be too tall to fit on the Letter sized paper. Be careful to understand the target output dimensions.

Another serious problem we have encountered is that sizing the graphic can disable it if the graphic does not fit on the page, or overruns margins. Visual FoxPro is kind enough to ignore the bad positioning and not trigger an error. On the other hand, it can be difficult to track down the problem when the graphic is not printed.

There are two ways we have used watermarks. One is a background image; the other is a foreground image. The background image can be used for large logos or to give the report a texture. These images are usually a faded, gray-scale image if printed on black and white laser printers and even on color printers. The foreground image is used as a way to "stamp" the report with a status such as draft, confidential, or top-secret. Putting it on top will cause the report contents to be overwritten or overshadowed, but the same is true with an actual stamp. Having these images in color also gives an added effect and can post added meaning to the content presented.

The watermark can be optionally not printed by using the Print When logic. This allows the users to indicate whether the report is a draft or the final version. Other times you will want to print the watermark each and every time, like when it is confidential.

How do I disable the report toolbar printer button? (*Example: **CreateReportFoxUser.prg***)

Reports typically have the requirement to be printed. Occasionally we have run across a report that needs preview capability without printing. Why? The report might contain sensitive information needed for review, but the boss only wants users with ultimate user security to be able to print it. Another situation is that the report might generate 500 pages of information, but the owner only wants that printed once by the supervisor and viewed by the rest of the staff. So how can we provide the application users with the standard preview mode and remove the printer button?

All the Visual FoxPro toolbar layouts (including the Print Preview) are stored in the Visual FoxPro resource file (FOXUSER.DBF/FPT). Hacking the resource file is not for the faint of heart. Fortunately there are some simple steps in creating a customized Print Preview toolbar and storing that layout in the resource file for use with your reports.

The first step is to create the resource file. There is some basic code you can run to create a clean file to start.

```
LOCAL lcOldSafety, ;
      lcOldResource, ;
      lcOriginalResourceFile, ;
      lcReportsResourceFile

lcReportsResourceFile = ADDBS(LOWER(FULLPATH(CURDIR()))) + "ReportsFoxUser.dbf"
lcOldResource         = SET("Resource")
lcOldSafety           = SET("Safety")

* Will used the specified resource file or
* will create a new one if one is not specified
SET RESOURCE ON
lcOriginalResourceFile = SYS(2005)
SET RESOURCE OFF
SET SAFETY OFF

USE (lcOriginalResourceFile) IN 0 SHARED ALIAS NotPureFoxUser
COPY TO (lcReportsResourceFile) ;
   FOR Id = "TTOOLBAR" AND ;
       INLIST(LOWER(Name), "report designer", "color palette", "layout", ;
                           "print preview", "report controls")

SET SAFETY &lcOldSafety
USE IN (SELECT("NotPureFoxUser"))

SET RESOURCE OFF
SET RESOURCE TO (lcReportsResourceFile)
SET RESOURCE ON

WAIT WINDOW "Modify the Print Preview Toolbar" NOWAIT NOCLEAR
SYS(1500, "_mvi_toolb", "_msm_view")

IF WEXIST("Print Preview")
   HIDE WINDOW "Print Preview"
ENDIF
```

```
MESSAGEBOX("Report resource file (" +lcReportsResourceFile+ ") generated...", ;
           0 + 64, ;
           _screen.Caption)

WAIT CLEAR
QUIT
```

You can follow along with the code. The first half of the code is setting up to copy certain records for the current resource file. If you are using one, the program will use it; otherwise, Visual FoxPro will create the default one when the **SET RESOURCE ON** is executed. The default resource file does contain the default Visual FoxPro toolbar definitions. It is important that we have the toolbar definitions. The **COPY TO** statement is specifying five toolbars. We have included all the toolbars used by the Report Designer. Why? Just in case you want to expose the capability of creating/modifying reports at runtime (see "How to allow end users to modify report layouts" on page 543 of *1001 Things You Wanted to Know About Visual FoxPro*, available from Hentzenwerke).

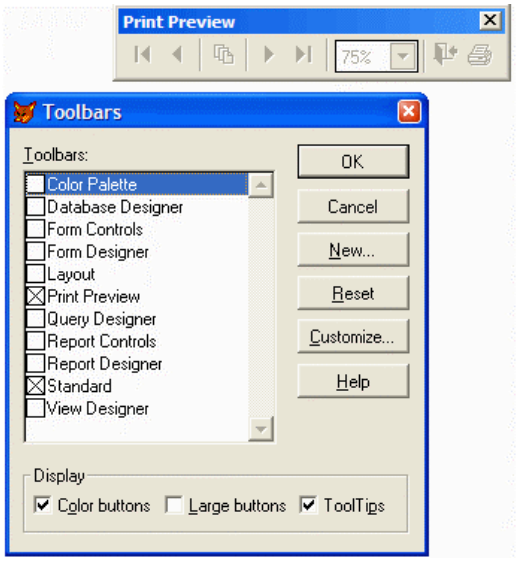


***Figure 3**. The Toolbar dialog provides a Customization feature.*

After the new resource file has been created we need to modify the toolbar. This has to happen manually. The program automatically opens up the Toolbar dialog (see **Figure 3**). You can manually get to this dialog via the View | Toolbars menu option. At this point you have to take control. Make sure to check on the Print Preview toolbar if it is not selected. Then press the Customize… button. The Customize Toolbar dialog is displayed. The Print Preview toolbar will also be displayed at this time if it previously was hidden. The key point here is to drag the printer icon off the toolbar and drop it anywhere but on the toolbar. This discards it from the toolbar definition. Close the Customize Toolbar dialog.

The program displays a message about the file being created as well as the name of the new resource file. The program then shuts down Visual FoxPro. You might think this is

extreme, but it is our experience that the definition of the toolbar remains in memory and is by definition saved to the current resource file. By shutting down Visual FoxPro, we are ensuring that the development resource file Print Preview toolbar remains intact.

Your customized resource file can be shipped separately with your custom EXE or you can compile it into the executable. If you deploy it separately, make sure you name it something other than FOXUSER.DBF just to make sure it does not get confused with the default resource file. To include it in the executable, add the resource file to the project as a free table. Mark it as an included file.

Now that you have the customized reporting resource file saved, you can temporarily use it as the application resource file. This can be used as the permanent resource file or just when printing the reports that do not need the print capability from the preview mode. This is accomplished with code as follows:

```
* NoPrinterFromPreview.prg
LOCAL lcOldResource, ;
      lcOriginalResourceFile, ;
      lcReportsResourceFile

lcReportsResourceFile  = "ReportsFoxUser.dbf"
lcOldResource          = SET("Resource")
lcOriginalResourceFile = SYS(2005)

SET RESOURCE ON
SET RESOURCE TO (lcReportsResourceFile)

* Make sure data is prepared
REPORT FORM WaterMarkDemo PREVIEW

SET RESOURCE TO (lcOriginalResourceFile)
SET RESOURCE &lcOldResource

RETURN
```

You can now incorporate this functionality into your own reporting mechanism. If you want to make sure that certain reports can be printed with the toolbar, you can leave in the default Visual FoxPro Print Preview toolbar. For those reports that should not be printed, have code like the preceding sample show the customized toolbar.

How do I detect if the user canceled printing and retain statistics for my reports? (*Example: CaptureReportDetail.prg, CaptureReportDetail.frx*)

We often run across requirements that indicate that a report needs to be tracked when printed and if it was successful. The same reports can be previewed, or printed to paper on an actual printer. How can you tell the difference and only how can you track it? Is there an easy way to track statistics on when the report was started, when it finished, and what printer it was directed to?

The key to determining whether the report was printed is checking for the existence of the Printing dialog that is displayed when Visual FoxPro prints a report to a printer. How can we check the existence of this window when the report is running? We can call a user-defined function. This function can check for the window using the **WEXIST()** function. The function can be called from any report expression. We recommend checking from the report summary

band or a group footer band based on the **EOF()** function. The reason we prefer this is that the user can cancel the report printing using the Cancel button on the Printing dialog. We want to log not just the fact that the user originally directed the output to a printer, but that they printed the entire report.

We have created an object with a number of properties to track various statistics.

```
poPrinterParameter = CREATEOBJECT("custom")
WITH poPrinterParameter
  .AddProperty("lStarted", .F.)
  .AddProperty("lPrinted", .F.)
  .AddProperty("cPrinter", SPACE(0))
  .AddProperty("mPrintInfo", SPACE(0))
  .AddProperty("tPreviewStarted", {/:})
  .AddProperty("tPrinterStarted", {/:})
  .AddProperty("tPrinterEnded", {/:})
  .AddProperty("nPages", 0)
  .AddProperty("cAlias", 0)
  .AddProperty("nRecords", 0)
  .AddProperty("tCompleted", {/:})

   REPORT FORM CaptureReportDetail NOCONSOLE TO PRINTER

  .tCompleted = DATETIME()
  GetPrinterInfo(poPrinterParameter)

  INSERT INTO ReportAudit ;
    (lStarted, lPrinted, cPrtr, mPrtrInfo, ;
     tPrwStart, tPrtrStart, tPrtrEnd, ;
     nPages, tCompleted) ;
    VALUES ;
     (.lStarted, .lPrinted, .cPrinter, .mPrintInfo, ;
      .tPreviewStarted, .tPrinterStarted, .tPrinterEnded, ;
      .nPages, .tCompleted)
ENDWITH
```

The report expressions will call a procedure and pass to the procedure the name of the band that the expression is in. For instance:

```
ReportStats("HEADER")
```

In the example report (CREATEREPORTDETAIL.FRX) we have calls to the ReportStat function in the title, report header, and **EOF()** group header and footer bands. The ReportStats method is where the printer parameter object properties get set:

```
FUNCTION ReportStats(tcBand)

tcBand = LOWER(tcBand)
?"Band=", tcBand, " ][ ", "Page=", TRANSFORM(_pageno)

DO CASE
  CASE INLIST(tcBand, "title", "bof")
    WITH poPrinterParameter
      .lStarted        = .T.
      .lPrinted        = WEXIST("Printing...")
```

```
      .cPrinter        = SET("Printer", 3)

      IF .lPrinted
        .tPrinterStarted = IIF(EMPTY(.tPrinterStarted), DATETIME(), ;
                                                        .tPrinterStarted)
      ELSE
        .tPreviewStarted = IIF(EMPTY(.tPreviewStarted), DATETIME(), ;
                                                        .tPreviewStarted)
      ENDIF

      .cAlias          = ALIAS()
      .nRecords        = RECCOUNT()
    ENDWITH

  CASE INLIST(tcBand, "header")
    WITH poPrinterParameter
      .nPages        = _pageno
    ENDWITH

  CASE INLIST(tcBand, "summary", "eof")
    WITH poPrinterParameter
      .lPrinted      = WEXIST("Printing...")
      .tPrinterEnded = IIF(.lPrinted, DATETIME(), {/:})
      .nPages        = _pageno
    ENDWITH
ENDCASE

RETURN SPACE(0)
```

The function is not only recording the various properties, but is displaying some interesting statistics on the Visual FoxPro desktop. This is not something you will want to do in a production application, but is showing us some interesting behavior. The band and the page number are displayed on the desktop. Ever wonder why the Report Designer was slow to display the last page of a long report when you are on the first page and why it is slow to show the second from last page of a long report when you are already on the last page? Visual FoxPro literally is running the pages through from page one. The reason we return a null string from the function is so nothing is printed on the report. Note that the expression is evaluated, calls the function, and prints the returned value on the report. If we returned `.T.`, a logical would be printed from the expression.

The final custom function called before inserting the recorded information into a table is GetPrinterInfo(). This procedure is concatenating the 13 return values from the `PRTINFO()` function. The `PRTINFO()` function provides the user or developer detailed information about the printer used to preview and print the report (see **Table 1**).

```
PROCEDURE GetPrinterInfo(toPrinterParameter)

#DEFINE ccCR   CHR(13)

IF VARTYPE(toPrinterParameter) = "O"
  IF VARTYPE(toPrinterParameter.mPrintInfo) # "U"
    FOR lnCounter = 1 TO 13
      toPrinterParameter.mPrintInfo = toPrinterParameter.mPrintInfo + ;
                                      TRANSFORM(lnCounter) + ") " + ;
                                      TRANSFORM(PRTINFO(lnCounter)) + ccCR
    ENDFOR
```

```
  ENDIF
ELSE
  * Nothing to provide
ENDIF

RETURN
```

***Table 1**. Listing of the various values that can be passed to the PRTINFO() function and what type of detail is returned.*

Printer setting	Information
1	Paper orientation
2	Paper size
3	Paper length (in .1mm increments)
4	Paper width (in .1mm increments)
5	Scaling factor
6	Number of copies to print
7	Default paper source
8	Resolution (negative value), or horizontal resolution DPI (positive value)
9	Color output
10	Duplex mode
11	Vertical resolution DPI
12	How TrueType fonts are printed
13	Collated printing

This solution does not differentiate if the report was printed to an Acrobat PDF file or a fax printer driver. You would need to understand all the different possible printer drivers to determine whether the report was printed on paper or to electronic media.

Crystal Reports

Crystal Reports is a 10-year-old product available from Crystal Decisions (formerly Seagate Software). It is recognized in the software development community as the most popular report creation, generation, presentation, and export tool. Visual FoxPro developers have successfully generated reports via the built in Report Designer for years and have complained that it has not been significantly enhanced to keep up with the latest reporting needs of business. Crystal Reports is a product Visual FoxPro developers can use to address these limitations.

There are three packages available: Standard, Professional, and Developer. The Standard edition only supports English. The Professional and Developer editions come in English, German, French, and Japanese. You can use the Standard edition to create almost any kind of report that Crystal Reports supports (standard form, cross-tab, labels, mail merge, sub-reports, top-n, and drill down). The Standard version includes the report expert/wizards, connects to most common data sources (including Visual FoxPro, FoxPro 2.x, Access, SQL-Server, dBASE, Clipper, Outlook, and so on), and exports to different file types (PDF, XLS, HTML, XML, RTF, and so forth). Each version comes with thorough documentation to support the features included.

The Professional edition includes all Standard edition features, plus Web reporting support (support for all the popular Web servers), more sophisticated SQL features, and additional data sources (Oracle, DB/2, Informix, Sybase, Microsoft Internet Information Server, Lotus Notes/Domino, and so on).

The Developer edition has all the Professional edition features, plus the report integration with custom applications (Report Designer Component, Automation Server, Print Engine API, ActiveX control, Delphi control, and MFC class libraries), customizable report preview window, drill down in the report preview window, Microsoft Transaction Server (MTS) support, and support for ActiveX Data Objects (ADO).

The Crystal Decisions Web site is at www.crystaldecisions.com. You can find plenty of information at this Web site to help determine which package of Crystal Reports will fit your needs, as well as information on training and support. The site also has a complete matrix of features with the editions that include the features. We have found the Web site a bit disorganized, but they do have published sales phone numbers as well if you cannot find enough clear information to make an informed decision.

Visual FoxPro developers with clients for which they distribute reporting as part of their custom applications will need the Developer edition. The royalty-free report preview control is a must-have for custom applications. Upgrading from any previous version of Crystal Reports (older license is included in Visual Studio 6) to the Developer Edition v8.5 will run you around US$260; if you need to purchase the full version you will have to spend US$495. These prices were taken directly from the Crystal Decisions Web site. You may find it cheaper at retail outlets on the Internet.

It is important to note that the discussion in this section refers to Visual FoxPro data in many instances. When we refer to data with Crystal Reports, this can be Visual FoxPro tables, views, or cursors generated from a remote view or SQL-passthrough command. The power of Visual FoxPro is that it can work with so many data sources with the same code we have been using over the years to manipulate and report.

The version of Crystal Reports we are using to discuss features and develop the examples for this chapter is Crystal Reports Developer v8.5 (version 9 was released just as our book entered the copy editing stage). The focus of the rest of the chapter is to demonstrate some of the features in Crystal Reports that directly resolve some of the limitations of the Visual FoxPro Report Designer.

Why should I consider Crystal Reports for reporting?

The reason you will use Crystal Reports instead of the Visual FoxPro Report Designer will depend on your customer reporting needs, and one or more of the weakness you find in the Visual FoxPro Report Designer.

Crystal Reports addresses a number of the limitations of the Visual FoxPro Report Designer. The features of Crystal Reports that are not in Visual FoxPro include a real preview zoom (up to 400%), data searching, drilldown capability from summary to detail data, sub-

reports, active hyperlinks, integrated graphing, document properties, and seamless data exports (Excel, Word, RTF, HTML, or PDF and retain all formatting). Crystal Reports also includes multi-line header bands, detail bands, and footer bands. You can easily add multiple lines to any section of the report and conditionally print those lines.

Visual FoxPro developers who create Web-based applications will appreciate the extensive Web reporting capability included in the Developer edition of Crystal Reports. These reports can be called directly from Active Server Pages.

Crystal Reports is also the standard reporting tool included in Visual Studio .NET so all your skills learned will be transportable if you also do development on the .NET platform. A special version of Crystal Reports is included in Visual Studio .NET.

It is important to note that you can have both Visual FoxPro reports and Crystal Reports integrated into the same application. There are definite concerns about a consistent user interface and users seeing functionality in the Crystal Reports and wanting that functionality in the other reports. The reason we mention this is that you do not have to convert all reports in an application to introduce Crystal into your deployed solutions.

What techniques can be used to integrate Visual FoxPro data with Crystal Reports?

Crystal Reports supports four different mechanisms to interact with Visual FoxPro 7 data: the native driver for FoxPro 2.x tables, the Visual FoxPro 6.0 Open Database Connectivity (ODBC) driver, the new Visual FoxPro 7.0 Object Linking and Embedding Database (OLE DB) driver and ActiveX Data Objects (ADO), and eXtensible Markup Language (XML). All four techniques work and are demonstrated in various examples in this chapter (see **Table 2**).

***Table 2**. The native datasources available for developers using various versions of FoxPro.*

Datasource	FoxPro 2.x	Visual FoxPro 6.x	Visual FoxPro 7.x
Fox 2.x Table	✓	✓	✓
ODBC		✓	✓
OLE DB			✓
XML			✓

The native driver for Fox2x tables is probably the most commonly used mechanism. The reason for this is that it was supported by previous versions of Crystal Reports and works with all versions of FoxPro, dating back to FoxPro 2.x. Visual FoxPro developers need to create the 2.x free tables using the `COPY TO...TYPE FOX2X` command. This is an additional step in the reporting process since the Visual FoxPro Report Designer will work with native tables, views, and SQL-Select cursors. The advantage of this technique is that Crystal Reports will read these 2.x tables natively and avoids the various layers of ODBC that can slow the reporting process down. There is nothing else that needs to be configured, such as an ODBC datasource. The disadvantage of this technique is that all the data types available in Visual FoxPro are not available in FoxPro 2.x tables (like currency, datetime, double, integer, character (binary), and memo (binary)). This means that some data will need translation, or loses some meaning altogether (like the time portion of datetime data). When using this

technique, our recommendation is to process the data into a cursor (either through the use of views or SQL-Selects) and to run the `COPY TO…TYPE FOX2X` before opening the report in Crystal Reports.

ODBC has been around for a long time and is a proven and reliable technology. To use Crystal Reports with an ODBC driver requires the use of the Visual FoxPro 6.0 driver. The advantage of this driver over the FoxPro 2.x tables is that you get direct access to the DBC and the long table names, views, and the long field names. There is no need for the extra processing to create the free table, and you no longer lose or need to translate data types not supported back in 2.x. The disadvantage is that you add the extra layer of the ODBC driver. If you are using the new DBC Events introduced in Visual FoxPro 7, the ODBC driver will not be able to read the database since the events alter the database version, which is unrecognized by the ODBC driver. You will also need to come up with a strategy of deploying the ODBC driver and creating the necessary datasource. These are not big hurdles, just things that need to be considered when deploying a solution. The ODBC technique can also be used to directly access back-end SQL data for applications that use Visual FoxPro for the user interface or business object tiers of an application.

The OLE DB/ADO technique takes advantage of the Visual FoxPro OLE DB provider that is new in Visual FoxPro 7.0. It supports the new DBC Events, and provides access to stored procedures, and the ability to create, modify, and delete functions. The OLE DB provider functionality supports the ability to execute stored procedure code independent of any Referential Integrity (RI) code, any default values, or column and row validation rules. The stored procedures return results in row sets. More importantly, it provides access to Crystal Reports via the OLE DB interface. The advantage of this interface is that you get access to tables, views, and stored procedures that return row sets, and all the data types provided by Visual FoxPro (unlike the native driver for 2.x).

Everyone is talking about XML these days. With the introduction of XML as a datasource in Crystal Reports v8.5, and the new `CURSORTOXML()` function introduced in Visual FoxPro 7.0, it is possible for developers to use XML formatted data in reporting solutions. The advantage is that the same exported data can be used for Web publishing, and it is remarkably fast creating the files considering the amount of information being written. The disadvantages include the fact that XML files can be quite large when a lot of data is used by the report, and the XML standards seem to change frequently and we are unsure what this could mean in regards to future implementations. We also find that the performance with large datasets is extremely poor. In our datasource comparison table (see Table 2), we noted that FoxPro 2.x and Visual FoxPro 6 cannot natively create XML. Developers using these versions can write their own XML creation code if so inclined. Visual FoxPro 6 developers can use the West Wind XML Converter (free wwXML class available from www.west-wind.com) to create the XML files.

What do I need to set up to run the samples in this chapter?

There are a number of samples generated in this chapter to demonstrate the capabilities of Crystal Reports and the mechanisms to get the data from a Visual FoxPro application to Crystal Reports. This section outlines the necessary items you will need to set up the ODBC

connections for the ODBC, XML, and OLE DB based reports. These ODBC connections can be established via the Windows ODBC Data Source Administrator, or directly from Crystal Reports using the Data Explorer. This dialog is presented when you create a new report in Crystal Reports. The process of setting up the datasources is identical whether you use the native Windows ODBC setup or Crystal Reports.

The ODBC driver is set up use the Visual FoxPro 6.0 ODBC driver v6.00.8167.00 (VFPODBC.DLL). One datasource is called "MegaFox7" and needs to be configured to the location that you load the chapter download. **Figure 4** shows how the driver was configured by the author and tech editor. The database container that is set is called MUSICCOLLECTION.DBC. When you create a new report, this datasource will be available on the Crystal Reports Data Explorer dialog, on the ODBC branch of the treeview.

Developers new to Fox development starting with version 7.0 might not have the Visual FoxPro 6 ODBC driver. It can be downloaded from Microsoft's Web site:
http://msdn.microsoft.com/vfoxpro/downloads/updates.asp.

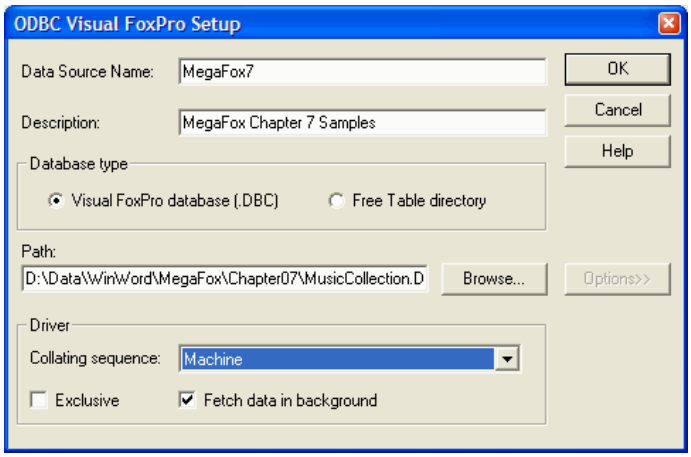


***Figure 4**. Set up the Visual FoxPro ODBC datasource MegaFox7 following the settings in this figure. Your directory structure will depend on where you loaded the chapter downloads.*

The second ODBC datasource is called "RandFoxODBC" and needs to be configured to the location that you load the chapter download. The database container that is set is called RANDOMTESTING.DBC. This datasource is only used by the RandExampleVFPODBC report.

The XML ODBC driver that ships with Crystal Reports was used for the XML based reports. The datasource is called "MegaFoxCrystalXML" and is based on a driver called CR XML v3.6, written by Merant, Inc. The version we used is 3.60.00.16 and the file name is CRXML15.DLL. **Figure 5** shows how the datasource should be configured for the samples in this chapter. On the General page we specified the datasource name, description, and the

location of the XML file. Make sure to remove the check on the "Require User Id and Password" on the Advanced page. Nothing needs to be set or filled in on the Options page. Make sure to test the connection if the file is already generated (the downloads ship with a sample RANDOM.XML file). When you create a new report, this datasource will be available on the Crystal Reports Data Explorer dialog, on the ODBC branch of the treeview.

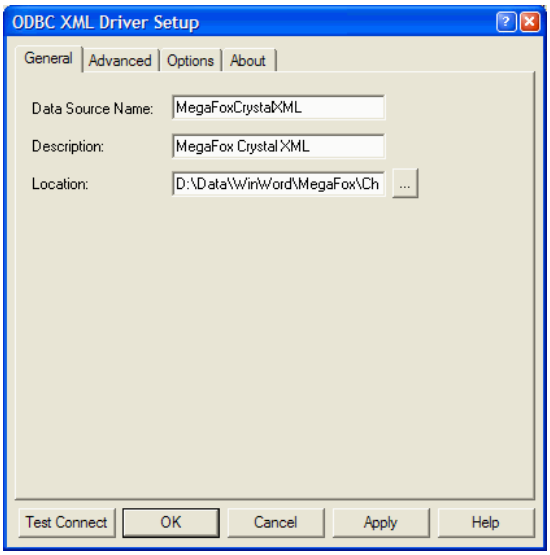


***Figure 5**. The Crystal Reports XML ODBC driver configuration for chapter samples.*

The OLE DB samples use the new Visual FoxPro OLE DB driver and are strictly code-based or configured in the actual report. If the report samples use the OLE DB interface, it is set up as follows:

1. Create a new report; you will be prompted for the datasource via the Data Explorer dialog.
2. Select More Data Sources and OLE DB in the treeview.
3. Double-click on the Make New Connection option.
4. The Data Link Properties dialog (see **Figure 6**) is displayed; select the Microsoft OLE DB Provider for Visual FoxPro, and press the Next button.
5. On the Connection page, enter in the database container (with path) and test the connection. You can also use the ellipsis button to select the database or a free table.
6. If the connection succeeds, you can proceed to the Advanced page and make selections you feel necessary.

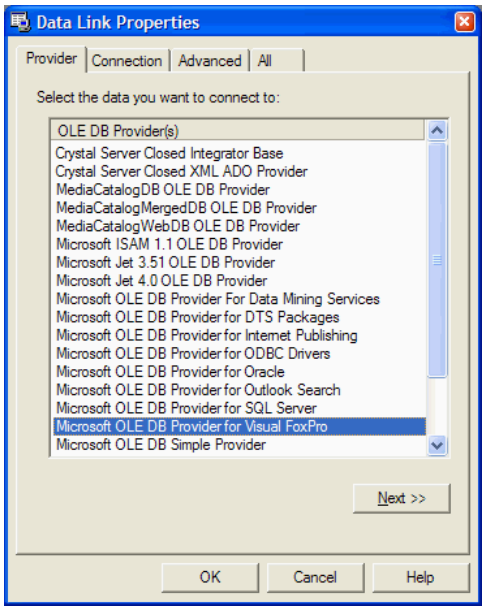


***Figure 6**. The Crystal Reports Data Link Properties dialog allows you to select the Visual FoxPro OLE DB driver if it is installed on the computer.*

The code used to make a connection to the Visual FoxPro OLE DB driver is generic:

```
loConn = CREATEOBJECT("ADODB.Connection")
loConn.ConnectionString = "provider=vfpoledb.1;data source=.\RandomTesting.dbc"
loConn.Open()
```

To create an ADO RecordSet we write the necessary SQL-Select that is executed over the connection:

```
loRS = loConn.Execute("select * from Random")
```

The Native FoxPro 2.x driver is configured in the same manner as the OLE DB driver. The only difference is that you select the Database Files in the Crystal Reports Data Explorer dialog, and then the Find Database File option. Make sure to select a FoxPro 2.x formatted table; otherwise, you will get an error that Crystal Reports cannot recognize the file.

All the files picked, whether it is a 2.x table, and ODBC connection, or an OLE DB datasource, will show up in the History branch of the treeview in the Data Explorer. This is a quick access point to previous data connections established.

Each of the report examples was developed in the Crystal Reports designer. This might seem obvious, but you will need Crystal Reports to open up the reports, look at the samples, and preview the reports with the data.

You might also run into a problem with the report samples in the chapter downloads depending on where you located the data. The datasource is stored in the report. To fix this issue in the samples, use the Database | Set Location… menu option (see **Figure 7**) to set

the location of the table. Click on the Set Location command button on the dialog to select the table or other datasource via the Crystal Report Data Explorer (see **Figure 8**). You can also try the Same As Report command button to fix the location. Our tech editor found a bug in Crystal Reports when changing the location of the data. If you do not change something on the report besides the data, Crystal will not recognize the change and will not save the new settings. Make sure to move a field back and forth before saving the report with the data change.

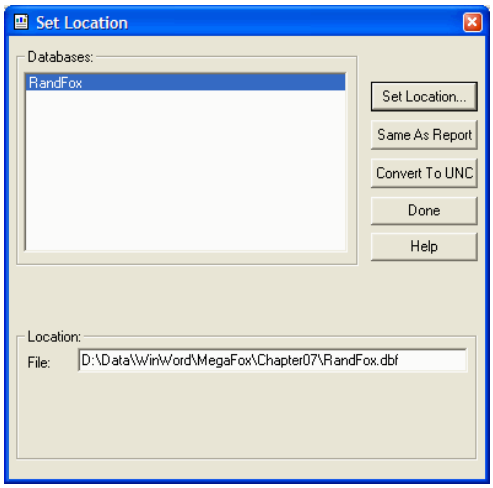


***Figure 7**. The Crystal Reports Set Location dialog will be useful to reset the location of the sample data used in the sample reports.*

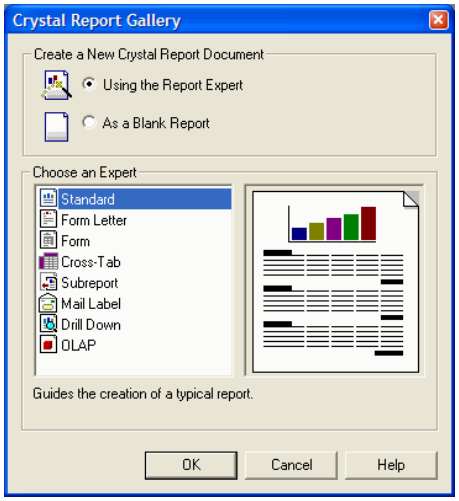


***Figure 8**. The Crystal Report Gallery is presented each time you begin a new report. You can either select an expert to guide you through the report creation process, or manually build a report by selecting the blank report option.*

What is the performance of the different techniques used to integrate Visual FoxPro data with Crystal Reports?

(*Example:* ***DataGenerator.prg, RandExampleFox2x.rpt, RandExampleXML.rpt, RandExampleVFPODBC.rpt, RandExampleVFPOLEDB.rpt***)

After seeing the various file format and techniques to integrate the data into a Crystal Report, a question immediately came to mind. Which combination would perform faster?

We set up the following test conditions in the DataGenerator program (included in the chapter downloads so you can also run it). The table we generated has at least one column for each of the common data types. We added two indexes on the table.

```
CREATE TABLE random ;
  (iKey i, ;
   cCharacter c(30), ;
   cHyperLink c(40), ;
   cMailTo c(40), ;
   yCurrency y, ;
   mMemo m, ;
   lLogical l, ;
   dDate d, ;
   tDateTime t, ;
   nNumeric n(13,3))

INDEX ON cCharacter TAG CharIndex ADDITIVE
INDEX ON yCurrency  TAG CurrIndex ADDITIVE
```

We populated the table with a different number of rows to test the report performance with various datasets. Each of the columns is filled with mostly random data with the following code:

```
lcTruth      = "VFP Rocks "
m.iKey       = 0

FOR i = 1 TO tnLoopCount
  IF m.iKey > 0 AND MOD(m.iKey, 100) = 0
    WAIT WINDOW "Processed " + TRANSFORM(i) + " of " + ;
                TRANSFORM(tnLoopCount) + "..." NOWAIT NOCLEAR
  ENDIF

  m.iKey       = m.iKey + 1

  DO CASE
     CASE MOD(m.iKey, 4) = 1
       m.cCharacter = "Geeks and Gurus"
       m.cHyperLink = "http://www.GeeksAndGurus.com"
       m.cMailTo    = "RASchummer@GeeksAndGurus.com"
     CASE MOD(m.iKey, 4) = 2
       m.cCharacter = "Tightline Computers"
       m.cHyperLink = ""
       m.cMailTo    = "AndyKr@CompuServe.com"
     CASE MOD(m.iKey, 4) = 3
       m.cCharacter = "Steve Dingle Solutions"
       m.cHyperLink = "http://www.SteveDingle.com"
       m.cMailTo    = "Steve@CompuServe.com"
     CASE MOD(m.iKey, 4) = 0
       m.cCharacter = "Hentzenwerke Publishing"
```

```
      m.cHyperLink = "http://www.hentzenwerke.com"
      m.cMailTo    = "Whil@Hentzenwerke.com"
  ENDCASE

  m.yCurrency  = NTOM(m.iKey * 3 + (INT(RAND()*100)/100))
  m.mMemo      = REPLICATE(lcTruth, RAND()* 10)
  m.lLogical   = IIF(MOD(m.iKey, 2) = 0, .F., .T.)
  m.dDate      = DATE() + m.iKey
  m.tDateTime  = DATETIME() + (RAND() * (100000 + m.iKey))
  m.nNumeric   = m.iKey + RAND()

  INSERT INTO Random FROM MEMVAR
ENDFOR
```

Once the table is filled with the random data, we need to create the files that Crystal Reports can read. We created a FoxPro 2.x free table using the `COPY TO` command, the ADO recordset was created by making a connection to the Visual FoxPro OLE DB driver and querying all the records in the random table, and the XML file was created with the new Visual FoxPro `CURSORTOXML()` function. The ODBC connection accesses the Visual FoxPro data directly; therefore no intermediate data needs to be created.

```
COPY TO RandFox.dbf WITH production TYPE FOX2X
loRS = loConn.Execute("select * from Random")
CURSORTOXML("curExport", "Random.XML", 1, 1+4+512, 0)
```

***Table 3**. The best creation time (seconds) to export the records from a Visual FoxPro cursor based on COPY TO, ADO Connection Execute SQL, and CURSORTOXML().*

	1000	10,000	25,000	50,000	100,000	250,000
Fox2x	0.130	1.202	2.994	5.868	16.423	51.143
ADO	0.191	0.260	0.551	8.042	26.458	137.177
XML	0.120	1.152	3.115	6.759	37.914	88.206

***Table 4**. The resulting file size (bytes) of exported files.*

	1000	10,000	25,000	50,000	100,000	250,000
Fox2x	265,362	2,651,458	6,633,426	13,261,122	26,524,962	66,298,754
ADO	N/A	N/A	N/A	N/A	N/A	N/A
XML	455,117	4,558,491	11,396,065	22,789,205	45,583,734	113,939,996

***Table 5**. The best time (seconds, pages in parentheses) it takes to show the Crystal Report in a Visual FoxPro form.*

	1000	10,000	25,000	50,000	100,000	250,000
Fox2x	<2 (15)	5 (146)	13 (363)	44 (725)	98 (1450)	136 (3624)
ODBC	<2 (15)	6 (146)	14 (363)	27 (725)	52 (1450)	163 (3624)
OLEDB	<2 (15)	5 (146)	10 (363)	26 (725)	37 (1450)	140 (3624)
XML	4 (18)	24 (176)	91 (439)	346 (878)	600+ (DNF)	330+ (GPF)

The times to show the Crystal Report in a Visual FoxPro form are estimated because the form displays per the normal Visual FoxPro event sequence, and then the Crystal Report viewer takes additional time to process the report. We used the Visual FoxPro clock in the status bar to manually start the tests and estimate the time it took to execute before the report was displayed (see **Tables 3-5**).

Your test timings might differ from ours and we encourage you to run the test program to analyze the results and make your own conclusions. The timings you see published in this book were made on a machine equipped with a Pentium 450MHz, 224MB RAM, 12GB hard drive, with Visual FoxPro, Crystal Reports, Microsoft Word XP, and the usual handful of system tray applications running.

How do I create a report in Crystal Reports? (*Example: CrystalOnTheFly.prg*)

The Crystal Reports documentation is very complete and we have no intention of duplicating the manual on how to create a report. The intent of this section is to introduce the three methods of creating a report: using the Report Expert (wizard style), the manual way using the report designer, and the programmatic method.

The Report Expert is a wizard-style interface started by choosing the File | New menu option and selecting an expert on the Crystal Report Gallery dialog (see Figure 8). The Report Expert can be run more than once for the same report, but the second and subsequent runs destroy the existing report. The Report Expert supports standard reports (what we are used to in Visual FoxPro), form letters (mail merge), forms (business forms), cross-tabs, subreports (reports within reports), mail labels, drill down, and OLAP. The steps that follow will depend on the type of report you select. We are not going to cover all the reports, but will give you a sneak peak at the process for the most common report, the standard report.

The steps in the wizard depend on the type of report selected on the Crystal Report Gallery. First you select an expert to run. The first step of any of the experts is to select the datasource for the report (for various datasource options, see the section “What techniques can be used to integrate Visual FoxPro data with Crystal Reports?” in this chapter). Once you select the datasource, you select the fields, and then determine the groupings. This is not a complicated process because the expert guides you through the necessary selections to generate the report (see **Figure 9** and **Figure 10**).

The steps between the field selection and the last step vary depending on the expert selected. The last step of each expert is the Style page (see **Figure 11**). This is where you determine the layout of the report. Each report type has custom predefined layouts selections. Once the report is generated you can modify the report, just like you can after you generate a quick report in Visual FoxPro.

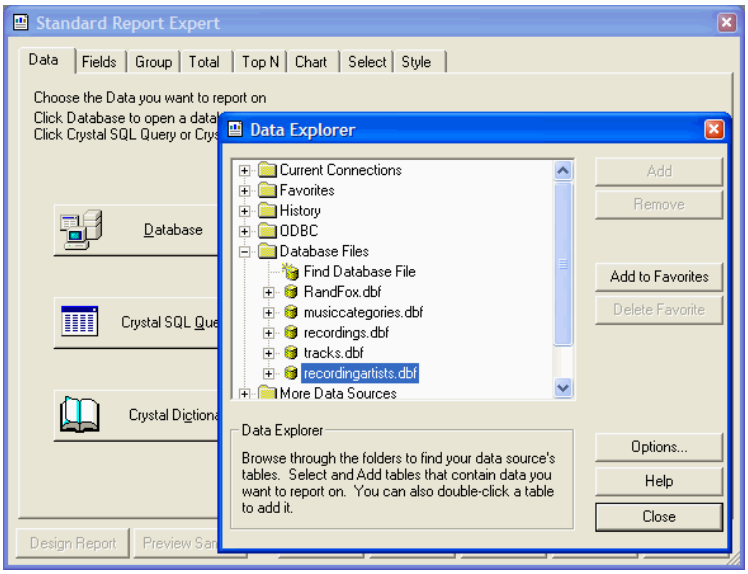


***Figure 9**. The first step of the Report Expert is to select the datasource and the associated table/views.*

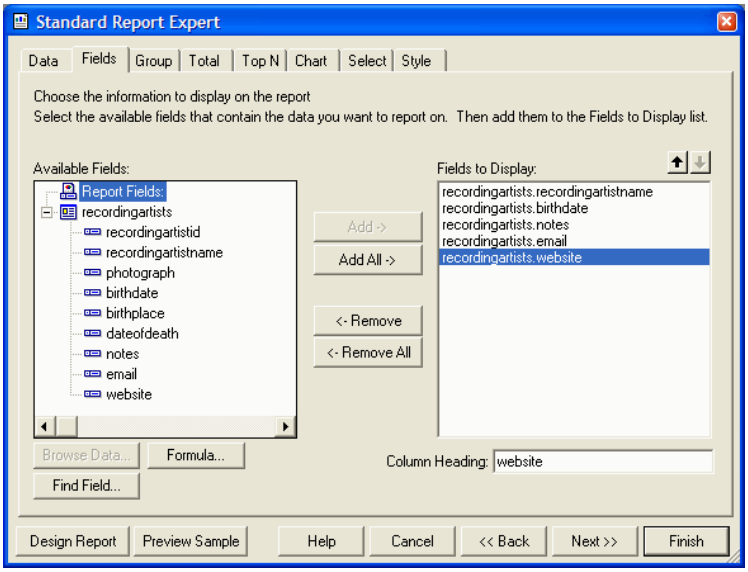


***Figure 10**. The Report Expert second step provides a dialog to select the fields.*

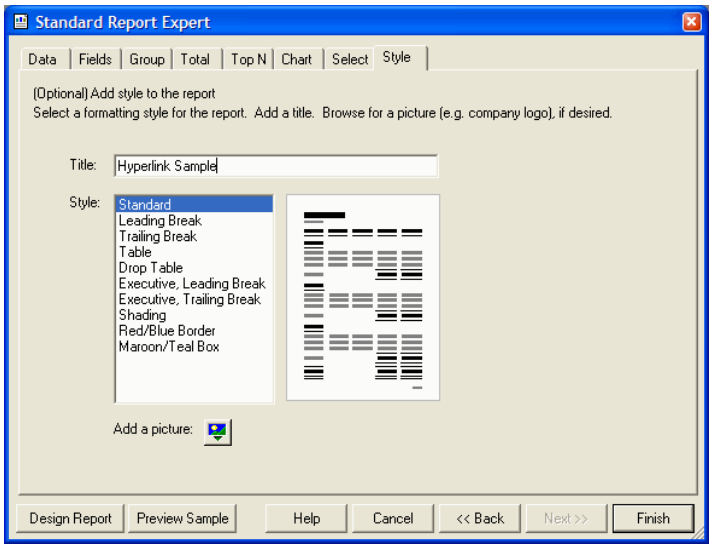


***Figure 11**. The last step in the Report Expert is where you determine the general layout of the report by picking one of the predefined styles.*

The second approach is the manual method. This is exactly like it sounds. Select the File | New option from the menu, select As a Blank Report in the Crystal Report Gallery dialog, select the datasource to use in the report (verify/adjust joins if necessary), begin adding fields to the report, set any groupings, add a chart, and so on. Everything on the report is added by the developer using the menu and toolbars in the report designer.

The last option to creating a report is the programmatic option. This is where the developer uses the exposed Crystal Reports Automation interface to create a report. The Crystal Reports 8.5 ActiveX Designer Run Time Library can be opened in the new Visual FoxPro Object Browser to start exploring the various properties and methods for the various classes. Here is some example code to get you started:

```
* CrystalOnTheFly.prg
LOCAL loCrystalReports  AS CrystalRuntime.Application, ;
      loReport , ;
      lcConnection, ;
      lcSql, ;
      lnColor, ;
      loReportObjects, ;
      loField, ;
      lcCrystalReportName, ;
      lcMessageCaption

lcConnection         = "Provider=vfpoledb.1;Data Source=.\MusicCollection.dbc"
lcSql                = "select * from recordingartists"
lcCrystalReportName  = "ReportOnTheFly.rpt"
lcMessageCaption     = "Crystal Report on the Fly!"

* Instantiate Crystal Runtime and add the report viewer to the form
loCrystalReports     = CREATEOBJECT("CrystalRuntime.Application")
loReport             = loCrystalReports.NewReport()
```

```
loReport.ReportTitle = "Crystal Report on the Fly"

* Add the OLEDB connection and specify table
loReport.Database.AddOLEDBSource(lcConnection, "recordingartists")

IF VARTYPE(loReport) = "O"
  WITH loReport

    WITH .Sections
      FOR i = 1 TO .Count
        ? .Item[i].Name

        IF MOD(5,i) = 2
          lnColor = RGB(0,255,0)
        ELSE
          lnColor = RGB(255,255,255)
        ENDIF

        .Item[i].BackColor = lnColor
        .Item[i].AddTextObject(.Item[i].Name, 0, 0)

        * Report Page Header
        IF LOWER(.Item[i].Name) = "section1"
          * Left and Top properties in Twips (~1441 per inch)
          .Item[i].AddSpecialVarFieldObject(10, 0, 200)    && crSVTReportTitle
          .Item[i].AddSpecialVarFieldObject(17, 5584,0)    && crSVTPageNofM

          loReportObjects = .Item[i].ReportObjects

          FOR j = 1 TO loReportObjects.Count
            * Right align the Page N of M object
            IF LOWER(loReportObjects.Item[j].Name) = "field2"
              loReportObjects.Item[j].HorAlignment = 3   && crRightAlign
            ENDIF
          ENDFOR
        ENDIF

        * Report Detail
        IF LOWER(.Item[i].Name) = "section5"
          loField =
            .Item[i].AddFieldObject("{recordingartists.recordingartistname}", ;
                                    1441, 0)
          loField = .Item[i].AddFieldObject("{recordingartists.email}", ;
                                            6000, 0)
        ENDIF
      ENDFOR
    ENDWITH

    .SaveAs(lcCrystalReportName, 2048)  && Saves file in v8 format
  ENDWITH
ELSE
  MESSAGEBOX("Crystal Reports could not generate a new report at this time.", ;
             0+16, lcMessageCaption)
ENDIF

* Crystal clean up
loReportObjects  = .NULL.
loField          = .NULL.
loCrystalReports = .NULL.
```

```
MESSAGEBOX("Crystal Report " + lcCrystalReportName + " was created.", ;
           0+64, lcMessageCaption)

RETURN
```

The example is definitely not all encompassing and is just scratching the surface as far as developing a sophisticated report. The intent is to demonstrate that Crystal Reports can be generated programmatically. You can also expose the report designer in your custom applications. We are not particularly fond of the ad hoc report idea (mostly because our customers are not experts with the application data models), so we have not explored this option in depth and will leave this as an exercise for the reader. There are examples in the Crystal Reports documentation.

What happens when I change the structure of source cursor for the report?

Crystal Reports stores information about the report datasource in the report file. This can be a problem if the file layout of the underlying data changes. The report is not broken if you add fields, but causes Crystal Reports to crash when you preview a report that uses a datasource that has fields deleted. The other problem is that the new fields will not show up in the Field Explorer. So how do we fix the report to recognize the changes?

The process must be initiated by the developer using the Database | Verify Database menu option. The files involved must be accessible. If they can be opened, Crystal Reports will verify if any changes were made. If there are none, a message indicating this is displayed. If changes have occurred, the Map Fields dialog is displayed (see **Figure 12**).

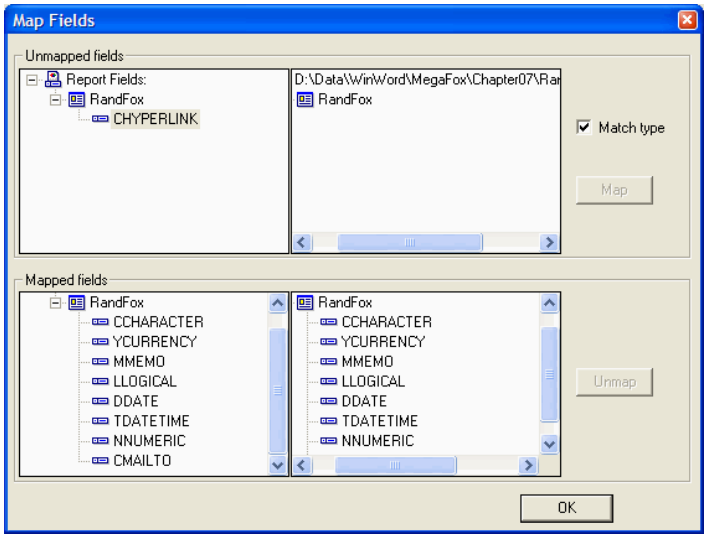


***Figure 12**. The Map Fields dialog is displayed when verifying the database if there are changes to the structures.*

The verification process will remove all objects on the report that were bound to fields that are no longer in the datasources. New fields will now show up in the Field Explorer and can be added to the report.

How do I implement hyperlinks in a report? (*Example: HyperlinkSample.rpt*)

One of the features we really wish the Visual FoxPro Report Designer supported is hyperlink capabilities. Today we store information related to the Internet such as e-mail addresses and Web sites. Crystal Reports' preview mode has live hyperlinks that will either navigate to the Web site URL (HTTP), initiate a message in the default MAPI compliant e-mail client (mailto:), open up a specified file, or run another Crystal Report.

The first step is to create a report and include the fields with the hyperlink data. In the report designer select one field with the hyperlink data and go to the main menu and select the Insert | Hyperlink option (optionally you can pick the Hyperlink button on the standard toolbar). This initiates the display of the Hyperlink page of the Format Editor dialog (see **Figure 13**). It is here that you specify the hyperlink settings. If you want the hyperlink to initiate the browsing of a Web page from a column in the table, make sure to check the "Current field value" option. If the column is an e-mail address, also check the "This field contains e-mail addresses" in the hyperlink information section. You can also hard-code hyperlinks and mailing addresses (better for one-time hyperlinks in headers and footers) by selecting "A web site on the Internet" or "An e-mail address." If you want to shell out a file to be opened you can select the "A file" option and specify the file to be opened. The default is to specify a file. You need to use the formula editor if you want to use the data in the field to open up the file.

One thing that we were pleasantly surprised by the first time we tested the capability is that the hyperlinks are carried over as live hyperlinks when the report is exported to an Acrobat PDF file.

Additionally, if the hyperlinks are for the Internet we change the color to navy blue and underline the text. This is handled through the font settings for the text objects. It adds visual clues to the end users that these are live hyperlinks. Otherwise, the users have to move the mouse over the object to see the mouse cursor change to get the hint.

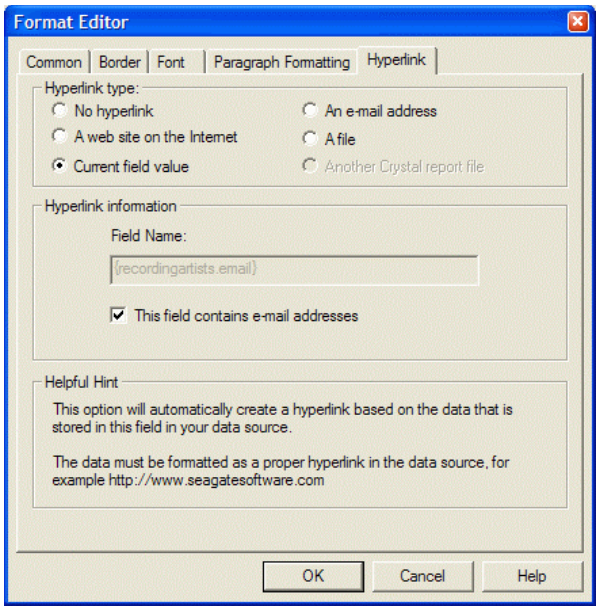


***Figure 13**. The Format Editor dialog provides hyperlink settings for developers.*

How do I display messages from within a report? (*Example: ReportAlert.rpt*)

Crystal Reports provides a feature called Report Alerts that allows developers to display messages to the users as they are viewing a report. This may alert the user to some specific data, a possible limit that was exceeded that they should look into, or bring attention to something important such as a stock that has finally reached a certain level.

There are three steps necessary to create a custom alert. Use the Report | Create Alerts menu option to start the process of setting the alert. On the Create Alert dialog press the New button to display the Create Alert dialog. You must enter the name of the alert, the message displayed to the user, and the condition that triggers the alert. The message can be based on a formula, which allows for the messages to be data driven.

The messages are only displayed the first time the condition is met (see **Figure 14**). So if you have a specific condition that displays an alert when the total sales are greater than one million dollars, and this condition is met as you scroll through the report, the message will not be redisplayed when you preview that page again. Once the data is refreshed (either by rerunning the report or by pressing the Refresh button on the toolbar, or the F5 key), the message will be displayed again when the condition is satisfied.

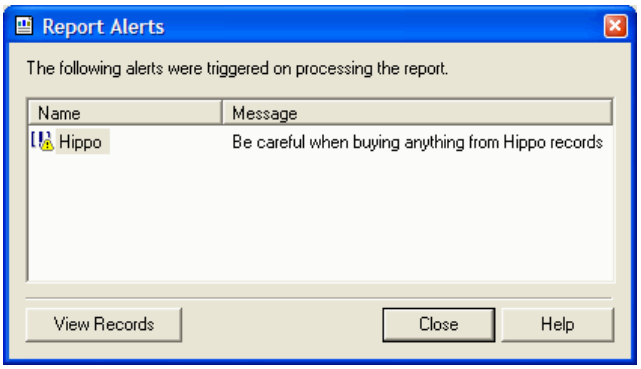


***Figure 14**. Report Alerts are presented in a dialog box when they are triggered.*

How do I add document properties to a report?

Document properties are fields that are predefined by Crystal Decisions, entered in the designer at development time, and stored in the report metadata file (RPT). The Document Properties dialog is displayed by selecting the File | Summary Info… menu option (see **Figure 15**).

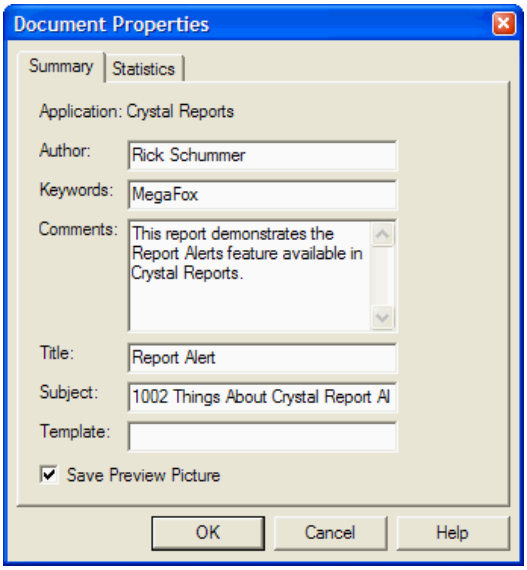


***Figure 15**. Document Properties is a feature we wish was in the Visual FoxPro Report Designer.*

The Author and Comment document properties can be printed on the report using the Special Fields. Only the first 256 characters of the Comment property can be printed on the

report. The properties also show up in the Windows XP Explorer ToolTip when you place the mouse over the file name. One thing we were disappointed with is that these properties are not carried over to the document properties when exporting to the PDF or the Microsoft Word file format, even though both of these products have the same feature.

How do I implement charts/graphs in a report? (*Example: GraphExample.rpt*)

One feature developers have begged for over the years is the ability to integrate charts and graphs into reports (see **Figure 16**). We have dedicated a whole chapter to graphing in this book because it is such a hot topic. Crystal Reports provides a simple, yet powerful implementation to include pie, bar, line, doughnut, and other types of charts right in your reports (see **Table 6**).

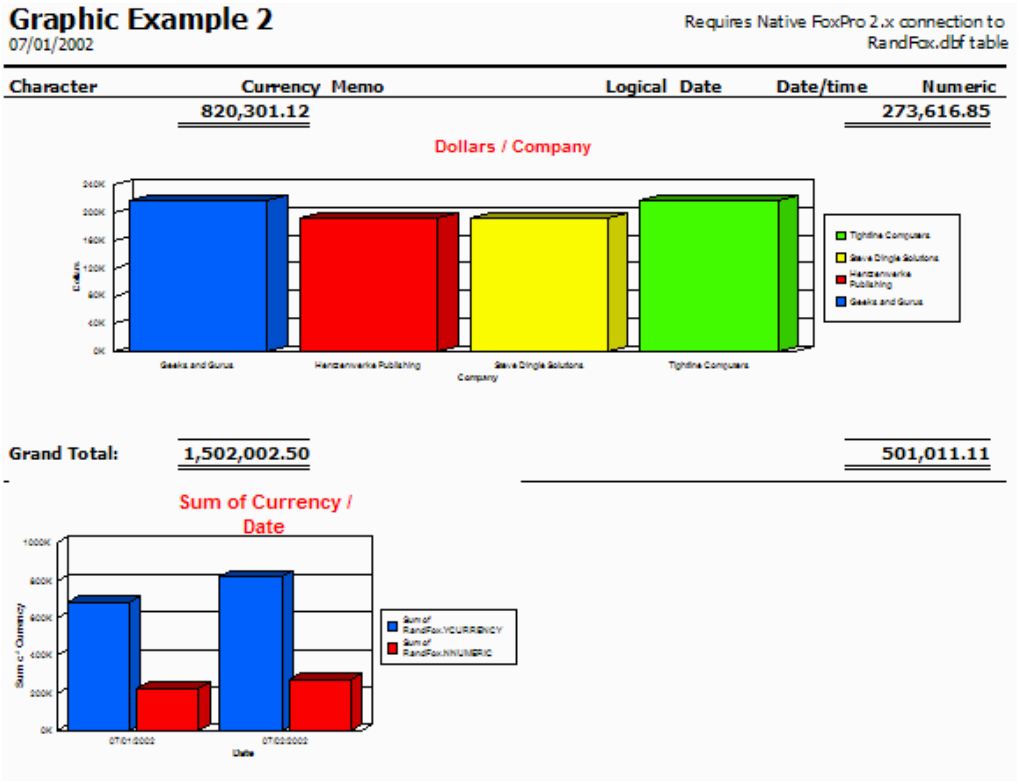


***Figure 16**. Multiple graphs can be incorporated directly with the other information presented in a report.*

***Table 6**. The different graph types available in Crystal Reports.*

Type	Variations/Formats	Vertical/Horizontal?
Bar	6	Yes
Line	6	Yes
Area	4	Yes
Pie	4	No
Doughnut	3	No
3D Riser	4	No
3D Surface	3	No
XY Scatter	1	No
Radar	2	No
Bubble	1	No
Stock	2	No

The process is started by using the Insert | Chart menu item, which starts the Chart Expert. There is no need to indicate where you want the chart because it will be inserted depending on the answers you provide to the expert. The expert seems to approach this a bit backwards in our opinion because you select the graph type before you specify the data to chart. The problem with this is that the chart types are dependent on the data configured and displayed in the report. We find ourselves switching between the Type and Data page frequently as we determine the best chart for the information and configure the chart.

On the Data page you can select Advanced, Group, Cross-tab, or OLAP. The Advanced option gives you the opportunity to configure all the data options. The Group option has the expert directing the chart into the report groupings and only lets you select data that is in the group headers or footers. The chart is also placed in the grouping selected. You can only access the Cross-tab or OLAP data options if you select a report of that type.

A nice feature is that the chart can be in the header or the footer. This means that the user can analyze the data in the chart and then proceed to review the details included. If the user wants to read the details first, then direct the chart to the footer (group, or report). You have complete control over the labels included on the chart via the Text page of the Chart Expert, except for the legend, which is data driven. The other labels default to the data values at first, but you have the capability to override them.

We have seen some powerful charting tools over the years, but not a clean implementation that is included right in the report. Crystal Reports provides a slick implementation and one we find has met the needs of our customer requirements. If not, you can resort to one of the various solutions presented in the charting chapter of this book.

How do I export reports to RTF, PDF, XML, and HTML formats?

One of the common features we are asked about by our customers and other developers is how to get the report data into a file like Microsoft Excel, Acrobat PDF, Rich Text Format (RTF), XML, HTML, and Microsoft Word. All of these formats can be created by Visual FoxPro developers using commands provided in Visual FoxPro, Automation, or a combination of third-party controls, but the kicker is that the users want the exported data to look exactly like the report they have loved using for years. Fear not, there are 21 different formats that a Crystal Report can be exported to.

There are several ways to initiate the export of the report. The Crystal Reports designer has a menu option File | Print | Export, as well as a toolbar button. This same toolbar button is available on the Report Designer Component (RDC), which is one way to display the reports in your runtime applications. The export process has several options. The first is to select the format of the export. This is where the user selects if they want a PDF, XLS, DOC, HTML, XML, or any of the numerous other export files created. The next selection is the destination. The destination will determine if the file is created to disk or is generated and opened up in the associated application. If you select Application, the host application will be started if necessary and the file displayed in the native format. If Disk file is selected the user will be prompted to name the file and select the location that it is written.

Some of the other formats that we are used to working with in Visual FoxPro via the `COPY TO` command include comma-separated values (CSV), character-separated values (comma-delimited), and tab-separated and are built-in exporting formats.

It is important to note that the reports do not always appear exactly as they do in Crystal Reports when exported to other applications. We have had good success with PDFs, HTML 4.0 (DHTML), and moderate to miserable experiences with Excel and Word. Your mileage may vary. We recommend trying different report layouts and testing the export of these layouts to see how good they really look in the native applications you need to export.

How do I implement drill down in my reports? (*Example: DrillDown.rpt*)

Drill down is a feature that allows users to view summary information, and if needed look at the underlying details that support the summaries without running a separate report. Visual FoxPro does not support this functionality in the Report Designer, but Crystal Reports does. The drill down feature is available on groups, charts, and maps and arguably could be the single biggest reason we looked at Crystal Reports in the first place.

Grouping natively supports drill down without any additional settings or special programming. It works by default. If you use the mouse to hover over a group header you will see it change to a magnifying glass (what Crystal refers to as the drill down cursor). Clicking on the object will spawn another view of the report (shown as an additional page on the report preview pageframe). If you have multiple grouping levels you can continue drilling down as deep as there are levels of the report. There are some considerations that you might want to review. The first is the ability to hide details and groups. This is accomplished by right-clicking on the group in the designer, and selecting the Hide (Drill-Down OK) option on the shortcut menu. You can reverse this by using the same shortcut menu and selecting Show. The other way to get to this setting is using the Section Expert (see **Figure 17**) available on the menu via the Format | Section option.

A typical implementation of this is to hide the detail band and any inner grouping bands (see **Figure 18**). This presents the user with a summary report. If they are satisfied with the summary information they can save browsing a lot of unnecessary pages of detailed data and get what they need. If they have a desire to see the details they can drill down into the next level. This could be another summary grouping, or all the way down to the details.

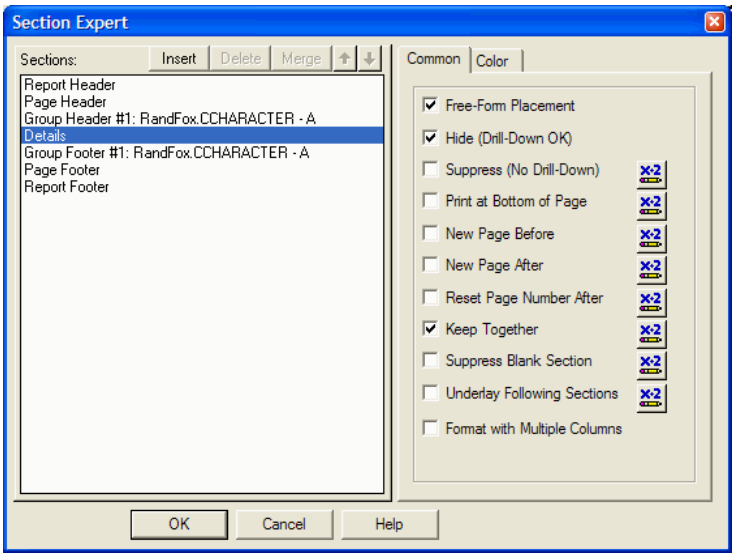


***Figure 17**. The Section Expert allows developers to hide the band and allow drill down, or suppress the capability from the users viewing the reports.*

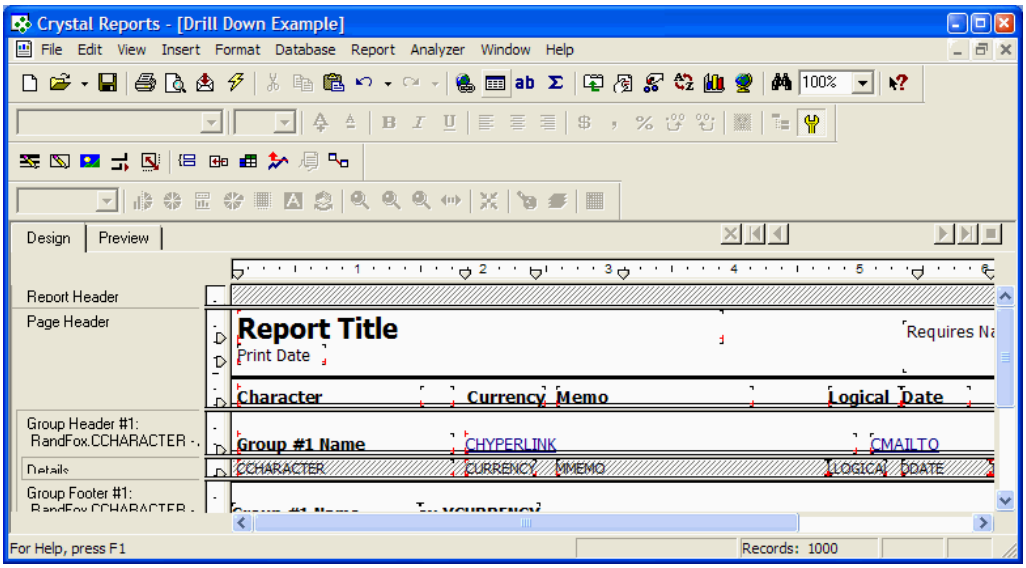


***Figure 18**. Bands that are hidden and allow drill down or are suppressed have a hash pattern when viewed in the Crystal Reports design page.*

If you do not want the users to be able to drill down in the report, you need to use the Suppress (No Drill-Down) grouping shortcut menu option (also found on the Section Expert). This will disable all drilling from that level and down to the innermost grouping or detail band.

Charting and Maps in group headers or footers work the same, but you have to right-click on the chart or map and select drilldown from the menu. We have not used this feature so we cannot address the advantages or disadvantages, but wanted to make sure you were aware that the feature is supported.

How do I work with subreports in Crystal Reports?

(*Example: SubreporUnlinkedtExample.rpt, SubreportLinkedtExample.rpt, SubreporUnlinkedtOndemandExample.rpt,, SubreportSub.rpt*)

Subreports are literally reports within a report. The reports can be related to each other or can be completely different data and format. The subreport is inserted into any section of the primary report and the entire report will print within the section it is inserted (see **Figure 19**).

The Crystal Reports documentation states that there are four situations where subreports are used. The first is reports that need to combine unrelated data (an example of this is the old problem we face in the Visual FoxPro Report Designer with the multiple detail line limitation). Subreports can address the issue of combining data that does not have a physical link in the table (calculated fields generated on the fly). The third idea is to present different views of the same data on the same report (weekly details and monthly summaries of the same sales information). The last idea is to present one-to-many lookups from a field that is not indexed.

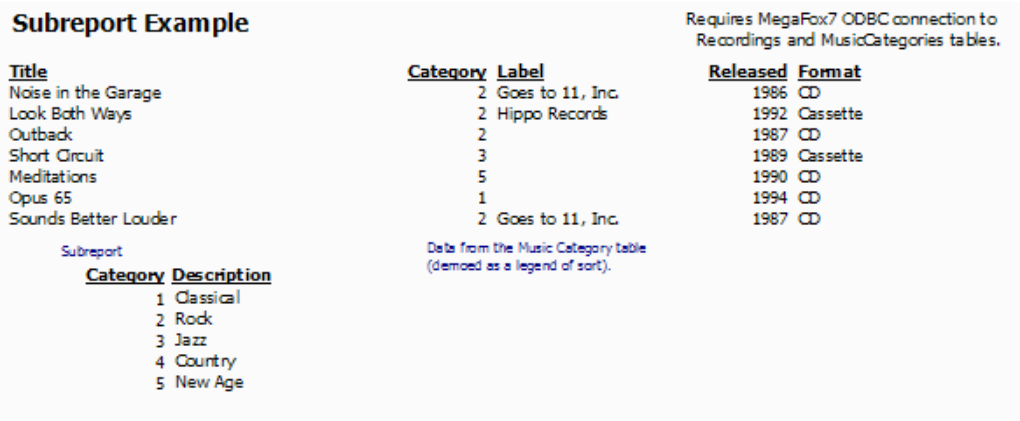


***Figure 19**. Subreports can be inserted into any report section. In this case, the unlinked subreport is positioned in the report footer.*

Subreports can be created with linked or unlinked data. The linked data is coordinated. Crystal Reports matches up records in the subreport with records in the primary report. For instance, if you create a primary report with base customer information and a subreport with location details, the data can be linked on the customer id. The primary report would first

display the base customer information, followed by the subreport with all in the location details. Unlinked subreports allow data to be completely disconnected. You could report on invoices in the primary report and inventory on the subreport and have them print on the same page of a report.

A subreport is created by creating a new report. When the Crystal Report Gallery is presented, select the Subreport option. If you are using the Report Expert you will follow the steps outlined by the Contained Subreport Expert. The steps are just like the standard reports. You select the primary report data, fields of the primary report, groupings, and the subreport. You have a choice to either select an existing subreport, or create a new one at this time. Creating a new subreport is just like creating a primary report. You select the datasource, the table, the fields, and so forth. Once the report is created or selected you need to determine whether you want the data linked or not (see **Figure 20**).

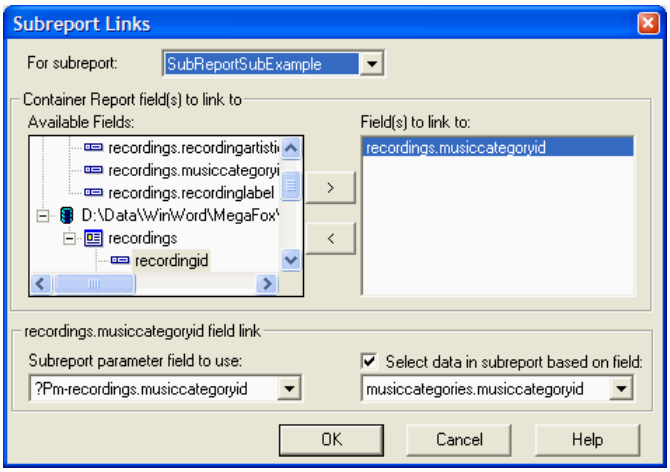


***Figure 20**. Linking data from a primary or container report to a subreport is much like joining tables via the Visual FoxPro View Designer. First select the field in the primary, then select the field in the subreport.*

In the section examples, in one report we linked the data and the other one we did not. The linked data example (SUBREPORTLINKEDEXAMPLE.RPT) was bringing in the music category description for the detail record. We know that we could just as easily include this in the based data via a join, but we wanted to demonstrate the linking. The unlinked subreport example (SUBREPORTUNLINKEDEXAMPLE.RPT) brings in all music category descriptions in a style of a legend.

One way to improve performance of reports with subreports is to mark the subreport as "on demand." This will print a hyperlink to the subreport instead of executing the necessary data queries to gather the information. Depending on the complexity of the subreport, a significant time savings when previewing a report can be gained. This feature is only useful for data that is rarely reviewed, but is occasionally important to be analyzed.

It might sound obvious, but a subreport is nothing more than a report, and it prints the entire subreport within the section that you have positioned it within the primary report. Therefore, if the recordset you are using for the subreport is rather large, there is a distinct possibility that the subreport could provide more information than the primary report. Subreports do have a limitation in that they cannot contain another subreport. You can run the subreport as a regular report and print the contents of the subreport.

This concept is not supported by the native Visual FoxPro reporting and takes some getting used to. We have found that working though a couple of examples helped us better understand the power of this functionality.

What can I do with the Report Designer Component? (*Example: AutomatePdfCreate.prg, GraphExample2.rpt*)

The Report Designer Component (RDC) provides developers with an ActiveX interface to different aspects of Crystal Reports. The RDC components include the Crystal Report Viewer (can be redistributed royalty-free), the Crystal Reports Print Engine (CRPE, royalty-free), and the Report Designer (specifically developed for Visual Basic developers, not a royalty-free distribution). We will discuss the Crystal Report Viewer in the next section, and concentrate on the Automation server capabilities in this section.

The fundamental answer to "What can I do with the Report Designer Component?" is, anything you can do with the Crystal Reports product. It is a comprehensive ActiveX interface to the various features in the report designer.

To demonstrate the capabilities, we will write code to print an existing report with charts and plenty of detailed information to an Acrobat PDF file. This is a common requirement for custom applications and one that developers have spent a lot of time solving. We dedicated an entire chapter in this book to Acrobat technology and several sections on how to automate this very process without user intervention. With Crystal Reports it boils down to 10 lines of code:

```
* Create the Crystal Report Object and open a report
loCrystalReports     = CREATEOBJECT("CrystalRuntime.Application")
loReport             = loCrystalReports.OpenReport("GraphExample2.rpt")

* Set the appropriate export options
loExportOptions                 = loReport.ExportOptions
loExportOptions.FormatType      = 31   && crEFTPortableDocFormat
loExportOptions.DestinationType = 1    && crEDTDiskFile
loExportOptions.DiskFileName    = "GraphExample2.pdf"

* Export the file without prompting the user
loReport.Export(.F.)

* Clean up object references
loExportOptions  = .NULL.
loReport         = .NULL.
loCrystalReports = .NULL.

RETURN
```

To be clear, you can create Acrobat PDF files directly from Crystal Reports. This option is available in the Crystal Reports Developer Edition and can be distributed royalty-free with your applications. This option can be much cheaper than your customers buying a license for Acrobat for each workstation, or you buying a third-party ActiveX control like Amyuni's PDF Creator.

In the section "How do I create a report in Crystal Reports?" in this chapter we automate the process of building a report programmatically. That example (CRYSTALONTHEFLY.PRG) shows how developers can literally use this interface to start with nothing and produce a report for the user. You can also manipulate the properties of a new or existing report to adjust things like datasources, sort orders, filtering, record selection, formatting objects, adding and changing groups, manipulating graphs, adding columns, changing the paper size, configuring alerts, working with subreports, changing export options, and much more.

The Visual FoxPro Object Browser reports that there are 432 properties, 297 methods, and 539 constants in the Crystal Reports 8.5 ActiveX Designer Design and Runtime Library (CRAXDDRT.DLL). The Crystal Reports documentation has plenty of examples that you will have to translate from Visual Basic 6.0 to Visual FoxPro, but we are all used to doing this with ActiveX vendors. The key thing to remember is that this is one ActiveX interface that works well with Visual FoxPro.

How do I work with the Crystal Report Viewer object? (*Example: frmCrystalPreview::CH07.vcx, frmCrystalPreviewTopLevel::CH07.vcx, PreviewCrystal.prg*)

Now that you have gone to all this work to learn about some of the advantages of using Crystal Reports in your application, the question begs, how do we display a report live in our applications? The answer is to use the Crystal Report Viewer object on a Visual FoxPro form.

The viewer is an ActiveX object that can reside in the base OleControl object. The sole purpose of this object is to display a Crystal Report and provide the features of the preview window. This is the same preview that you have available in the Crystal Reports when you want to preview the report as you develop it. The Crystal Reports Viewer has a toolbar to close any drill down pages (the first icon, the red X), print to a printer, refresh the data in the report (lightning bolt), toggle the group tree, really zoom the report (25% to 400%, with fit to page and fit to width options), and navigate to different pages via VCR buttons and the ability to specify a page. The last two buttons are the stop loading button, which is the way a user can stop a report from processing the data and generating the report, and the search button for users to search for text in the report.

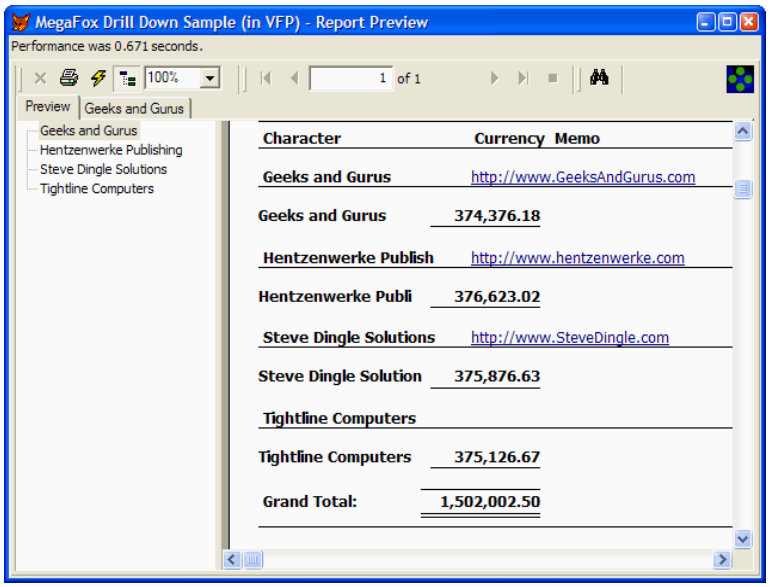


***Figure 21**. You can view a Crystal Report directly on a Visual FoxPro form.*

There is not a lot of code to display a report in the Visual FoxPro form (see **Figure 21**). The variable declarations for the Crystal Report Runtime application and the Crystal Reports Viewer are never used in the code. We declare them so we can use the references in the rest of the code to provide the power of IntelliSense to us during development. The Crystal Report application object is instantiated so we can open a report and assign that report to the viewer. The viewer is added to the form programmatically and sized to the form. The report object reference obtained from the application object is assigned to the custom *ReportSource* property and the viewer is instructed to view the report via the *ViewReport()* method. This code is all processed in the form *Init()* method.

```
LOCAL loCR  AS CrystalRuntime.Application, ;
      loCRV AS crViewer.crViewer

WITH this
  * Instantiate Crystal Runtime and add the report viewer to the form
  .oCrystalReports = CREATEOBJECT("CrystalRuntime.Application")
  .oReport         = .oCrystalReports.OpenReport(this.cReportName)

  .AddObject("oleCrystalReportViewer", "oleControl", "crViewer.crViewer")
  .WindowState = 2

  WITH .oleCrystalReportViewer
    * Set report viewer properties
    .Top                  = 20
    .Left                 = 1
    .Height               = this.Height - 2 - .Top
    .Width                = this.Width - 2
    .EnableProgressControl = .T.
    .ReportSource         = this.oReport
```

```
    IF this.lStartLastPage
      .ShowLastPage()
    ENDIF

    .ViewReport()
  ENDWITH
ENDWITH
```

It is important to point out that the report viewer will trigger an error if you try to display the viewer before the report is done loading. So we added this code to the form *Error()* event method.

```
* Form.Error() method
LPARAMETERS tnError, tcMethod, tnLine

IF tnError != 1440
   * Error 1440 is caused by the Crystal Report Viewer when you try to
   * display it before the report is done loading. (Thx to Craig Berntson)
   DODEFAULT(tnError, tcMethod, tnLine)
ENDIF

RETURN
```

You have a significant amount of control over the configuration of the viewer. You can control the display and enable/disable of the close button, drill down, group tree, navigation controls, print button, progress control, refresh button, search control, stop button, animation control, zoom control, or the entire toolbar. There are a number of related events that you can write code to execute as the users interact with the viewer control. Each of the navigation buttons has a Click event, as does the print button, export button, refresh button, and search button. There is an event that triggers when the zoom is changed, and when a drill down is performed on the detail, group, graph, or subreport. These are all well documented both in the control (using the Visual FoxPro 7.0 Object Browser) and in the printed and online documentation.

There are some quirks we found when developing this form and using it. The report is loaded after the form is displayed and the viewer control does not size to the form correctly when loading. We resize it in the form *Activate()* method that is fired after the report is done loading. We also developed a sample Top-Level form for those developers who have implemented Top-Level based applications.

What do I need to add to my deployment package when using Crystal Reports?

The files necessary for a Crystal Report deployment will depend entirely on the type of features you include in your applications. There are four categories of files to be concerned with: the Crystal Report Engine, Database Access, Exporting, and Additional Components. This can be a complicated process, but it is all detailed in the Help files called RUNTIME.HLP and LICENSE.HLP, which are located in the Developer Files\Help directory under the Crystal Reports root directory.

The Crystal Report Engine provides a number of options depending on the method of displaying the reports. Most developers using the current recommended way of reporting using

Crystal will be deploying the Report Designer Component (CRAXDRT.DLL). If you are using one of the other methods, check the RUNTIME.HLP file for the correct DLL to deploy.

The Data Access options are plentiful. If you have standardized on one or two mechanisms for integrating data with your reports you can quickly select the proper files to deploy. The categories include Direct Access Databases, ODBC Data Sources, Active Data and Crystal Data Object, OLE DB Data Sources – ADO, Crystal SQL Designer Files, and Crystal Dictionaries. Visual FoxPro developers will be particularly interested in P2BXBSE.DLL (located in the \Windows\Crystal directory) file because it supports direct access to FoxPro 2.x tables. Those implementing the OLE DB drivers should consider deploying the Microsoft Data Access (MDAC) as well as the VFPOLEDB.DLL file. Additionally you will have to consider how to create ODBC datasources on the users' computers.

The latest version of Microsoft Data Access (MDAC) files can be downloaded from http://microsoft.com/data/. The latest version does not include the Visual FoxPro ODBC driver that shipped with Visual FoxPro 6.0, nor the Visual FoxPro 7.0 OLE DB driver. You will have to ship these separately with your deployment package. Separate Merge Modules exist for the MDAC, the Visual FoxPro ODBC, and Visual FoxPro OLE DB drivers if you are using InstallShield Express to build your deployment packages.

Like the Data Access, there are a number of categories for exporting your Crystal Reports. We recommend distributing all the various royalty-free formats just to give your users all the options supplied by Crystal Decisions, unless you have a concern with the performance of a specific format, the application has requirements to exclude a format, or the users have security concerns with a format being accessible.

The final category is the Additional Components. This category includes Charting, HTML, Paged Range Export, SQL Expressions, and User Function Libraries. If you are using these features be sure to look into the files needed.

Developers who deploy applications with any language other than English will have additional considerations. There are resource files (same concept as the Visual FoxPro runtime resource files) that replace the corresponding English files.

If you use InstallShield Express v3.5 SP4, the process is simplified a bit (this is not the same product that ships with Visual FoxPro 7.0). You still need to understand the features you have implemented, but InstallShield Express has a wizard to step you through the selection process and will deploy the proper merge modules based on your selections. This can potentially reduce the size of the deployment package. The wizard is initialized when you select the Crystal Reports merge module within the InstallShield Express project.

All in all, the DLLs to be selected and deployed are numerous. The first time is the most difficult since you have to wade through all the options. Once you have developed and deployed your first application you will have a better understanding of what is necessary based on the functionality you have integrated.

The actual Crystal Report report files (RPT) can be distributed separately from your Visual FoxPro application. It is our experience that this is the most efficient way since we use the Crystal Report Viewing object with our applications. This object (discussed in the section "How do I work with the Crystal Report Viewer object?") opens up a report on the hard drive. You can write code that will copy the RPT file from your application to the drive and have the

viewer show the report. This is an extra step you will need to consider when this technique of deployment is desired.

Crystal Report wrapper objects for commercial frameworks

We wanted to spend some time developing a basic Crystal Report wrapper class, but shipping is a feature and we ran out of time before this book went to print. We do want to point to a couple of wrapper classes available for commercial frameworks in case you are working with one of these frameworks. Looking at these solutions can save you a lot of time.

CrystalVFE is available from F1 Technologies (www.F1Tech.com) for developers who use Visual FoxExpress. This product was developed by a Visual FoxExpress developer, Randy McAtee. The product was still in beta as of this writing, but it looks promising since it integrates into the Visual FoxExpress wizards and development interface. The wizards assist you in creating the Crystal Report and the classes involved support many of the Crystal Report features including subreports. Once the wizards generate the report and supporting user interface, you are free to modify the reports to your requirements. The mechanism to integrate the VFE business objects and the supporting data is through ADO recordsets. A Help file and the necessary classes are provided for developers to get a quick start to integrating Crystal Reports into their VFE applications. This is a commercial product, so developers are going to have to pay $399 to purchase this product.

Developers using the Mere Mortals framework by Oak Leaf Enterprises have Paul Mrozowski to thank for his free KAMMReport class library. This is a set of classes that support a user interface and a reporting engine. The reporting engine not only supports Crystal Reports, but also handles standard Visual FoxPro reports. There are classes to display the reports within your Visual FoxPro application. The Help file provides documentation for the classes, a "how to" section, and samples. The class library can be downloaded from www.KirtlandSys.com.

Visual MaxFrame Professional does not have any native support, but the open architecture provides plenty of hooks to override the support for the Visual FoxPro Report Designer. The report object has a method call for the `REPORT FORM` that can be easily overridden with the necessary calls to Crystal Reports.

What might you miss about the Visual FoxPro Report Designer when working with Crystal Reports?

Believe it or not, Crystal is missing some things that we have been accustomed to using for years with the Visual FoxPro Report Designer.

The biggest thing you will miss is the power of the FoxPro language being integrated into the expressions, groupings, and Print When conditions. Crystal Reports has a new Basic-like language, but it is not the same, or as mature, or as powerful as Visual FoxPro's language. To work around this you will need to build your cursors in advance and use the FoxPro language in the code that creates the record set used by Crystal Reports. This is a technique we have recommended for years when working with the Report Designer; now it is even more appropriate to adopt this approach. The Crystal syntax and the Basic-like syntax are easy enough to learn and have plenty of functions; it is just different from the FoxPro language we all love.

Conclusion

Reporting is still the cornerstone of custom business applications when it comes to users analyzing the information entered and generated by their applications. The Visual FoxPro Report Designer and Crystal Reports both serve Visual FoxPro developers well, and both provide developers with the tools necessary to present information in a valuable way for our customers in their custom applications. There is so much more that can be written on this topic for both report designers. Hentzenwerke Publishing has recognized this and has published *The Visual FoxPro Report Writer: Pushing it to the Limit and Beyond*, by Cathy Pountney, and has a book dedicated to Crystal Reports in the works.

Chapter 8
Integrating PDF Technology

The Adobe Acrobat Portable Document Format (PDF) is proven technology that allows Visual FoxPro developers to enhance the output generated by their custom applications. This chapter will show how you can integrate PDFs, extend the presentation of Visual FoxPro reports, and allow users to input data through PDF files into a Visual FoxPro application.

Generating Acrobat Portable Document Format (PDF) files has become commonplace and is as simple as printing output to a printer. If your customers are anything like our customers, they are asking for more and more integration of PDF output with custom applications. The Adobe Acrobat Web site has a quote on it that we think bests describe the Acrobat technology:

"Adobe® Portable Document Format (PDF) is the open de facto standard for electronic document distribution worldwide. Adobe PDF is a universal file format that preserves all the fonts, formatting, graphics, and color of any source document, regardless of the application and platform used to create it. Adobe PDF files are compact and can be shared, viewed, navigated, and printed exactly as intended by anyone with free Adobe Acrobat® Reader® software. You can convert any document to Adobe PDF using Adobe Acrobat 5.0 software."

Adobe PDF files can be published and distributed anywhere: in print, attached to e-mail, posted on Internet sites, distributed on CD-ROM, viewed on a Palm or Pocket PC device, or even displayed in a Visual FoxPro application using an ActiveX control provided by Adobe. In a nutshell, any information that can be printed to a Windows printer can be generated into a PDF file. The PDF files are typically smaller than their source files, and can be downloaded a page at a time for fast display on the Web.

PDF files also provide an alternative way of sharing documents and application output over a broad range of hardware and software platforms without sacrificing any formatting that can be lost using HTML.

Which version of Acrobat do I need?

Acrobat comes in three flavors: Reader, Approval, and the full-featured (known as plain old Acrobat). Adobe Acrobat was at version 5.0 when this book was written.

Reader is available free of charge and can be downloaded from Adobe's Web site. The generated PDF file can be viewed by anyone who has the Adobe Acrobat Reader. The Adobe Acrobat Reader displays the PDF file for viewing and has a number of features that include printing of the document, searching for text, and e-mailing the file to someone else. Users who just view the output generated by a custom Visual FoxPro application in PDF format can use this flavor of the product. Acrobat Forms can also be submitted to a Web process using the Reader version of the product.

You need the full-featured Acrobat application to be able to create PDF files, create Acrobat Forms, write JavaScript within a PDF, add electronic comments, or convert Web pages to PDF. Custom applications developed with Visual FoxPro that create a PDF file using an Adobe product will need the full version of Acrobat.

An individual Acrobat license is required for every workstation that will generate PDF files from your custom application. This means if you have 50 users working at 50 different workstations that access PDF generation functionality in the application, your customer will need 50 licenses at approximately US$225.

Acrobat Approval is available to save Acrobat Forms, apply e-signatures, spell check contents of a PDF, and secure documents so others cannot make changes. If users are entering data into an Acrobat Form and need to save this data to the server or workstation hard drive, they can use this version of the product. Using Acrobat Approval can provide significant deployment savings if generating PDF files is not a feature that is required, but form data needs to be saved.

What is needed to generate a PDF file?

Acrobat PDF files are generated via a printer driver loaded on the client PC. These are printer drivers just like ones for a laser or color printer. These print drivers have the intelligence to generate files in the PDF format. As noted before, these files retain all the needed information to duplicate the output exactly as the original application intended it to be printed.

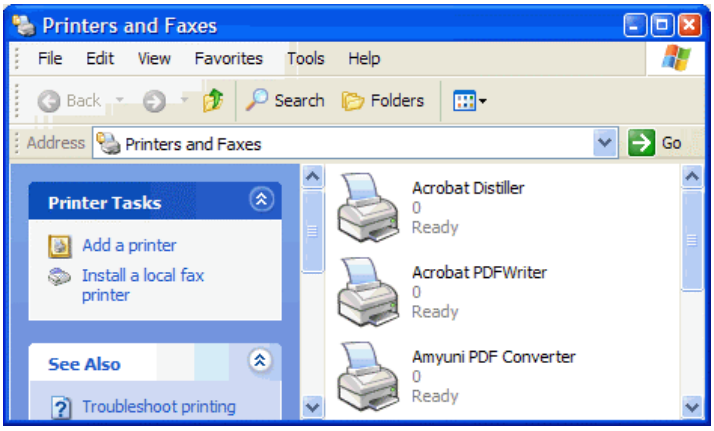


***Figure 1**. These are the printer drivers loaded when Acrobat and Amyuni drivers are installed.*

You can purchase the Acrobat product around US$225. When you install Acrobat (not the Reader) you get two printer drivers loaded (see **Figure 1**). The PDFWriter is an older, less sophisticated driver. Distiller is the more powerful and more current driver. We have had good success with the PDFWriter and find the limited features more than sufficient for our implementations. We have also found that it is faster in performance, which is good if the tradeoff of functionality is not limiting.

If you plan to use the Acrobat PDFWriter driver, you need to know that it is not loaded by default when installing Acrobat 5.0. You will need to select the custom setup and make sure to pick the PDFWriter to be installed.

One alternative to Acrobat that we have used successfully is the Amyuni PDF Converter (PDF Compatible Printer Driver). This runs $129 for a single-user license for one platform and $189 for all the Windows platforms (3.1, 95, 98, Me, NT, 2000, and XP). The Developer license contains the ActiveX interface and is purchased one time ($800 for single OS platform, $1150 for all platforms) and has a royalty-free distribution. The Developer license only allows features to be accessed via the ActiveX interface and does not have any user interface, and no permanently loaded printer driver. This works well for Visual FoxPro (and other Visual Studio tools) based applications. The printer driver only exists at the time the driver is used and is generated on-the-fly when the ActiveX control is accessed to generate the PDF file. If your users need the user interface to the PDF Converter, they can get a site license for $2500 for a single OS platform or $3600 for all OS platforms. There is a new Professional version with encryption and Web optimization available.

Okay, this sounds good so far, but wait—there is more! Amyuni also has Visual FoxPro specific examples to boot and they actually advertise in Visual FoxPro periodicals! There is even more; they have even gone as far as developing an FLL API file for use with Visual FoxPro. Now, the FLL solution is not always recommended since the ActiveX interface works well (unless you need bookmarks), but it is nice that Amyuni is showing support for Visual FoxPro in this fashion.

We are not trying to include an ad here for Amyuni, just trying to provide a baseline so you can evaluate the advantage or disadvantage of this product line. We advise you to check out the Amyuni.com Web site for all the details.

How do I determine which PDF product to license?

All PDF creation features are available in both the Adobe PDFWriter/Distiller and Amyuni PDF Converter drivers. The Amyuni PDF Converter gives an unlimited distribution product with the Developer license. You or your client will need to purchase a full copy of Acrobat for every PC that will generate PDF files. In a small company (fewer than six users), it may be better to go the Acrobat route; larger sites or vertical market apps should seriously look at the Amyuni product. Adobe does have an Open Options Site License Program for organizations with 1,000 or more workstations. Contact Adobe for more specifics. Acrobat 5.0 also has the interactive development environment as well, which may be something you or your customers will need.

Once the Acrobat printer driver is loaded it automatically becomes available to all Windows applications and is actively visible in several applications already installed. For instance, all the Microsoft Office (v97, 2000, and XP) applications have the PDFMaker macro/toolbar installed and available. The Amyuni version will not be available to other applications unless you get the site license.

There are other PDF writers available that are similar in functionality and implementation. We are most familiar with the Amyuni product, which is why we have chosen it for discussion

in this chapter. We are not endorsing this product over the others, just trying to express implementation ideas for these tools.

How can I use PDF technology in my Visual FoxPro apps?

An example of the use of these components is the company accountant publishing the sales results tracked in a custom database application (naturally developed by a top gun Visual FoxPro developer) to a PDF file. This file could be transferred via e-mail to the sales force and they could view it on their laptops for review. Changes can be e-mailed back to the accountant and updated in the database. The accountant re-creates the PDF file and posts it on the company Web site. Now all employees in the company can hit the company Web site to see how well the company sales are going.

So why publish to the PDF format instead of HyperText Markup Language (HTML) format? HTML was designed for single-page documents with limited formatting capabilities. The presentation of the document differs from one computer to another and from one Web browser to another. Also, to transmit a single page, one needs to transmit many files containing different parts of the page (one file for each graphic). PDF documents can have hundreds of pages contained in one file with all the formatting capabilities that modern applications provide.

How do I output Visual FoxPro reports to PDF using Adobe Acrobat? (*Example: PromptPDF.prg*)

Once the full version of Adobe Acrobat is installed, generating Visual FoxPro reports to a PDF file is quite simple. First you make sure that the PDF Printer Driver is set as the default printer for the Visual FoxPro application. This can be any Visual FoxPro report. If the report has a hard-coded printer driver in the TAG, TAG2, and EXPR fields for a printer other than the Acrobat driver, the following code does not work. No special driver setting has to be made in advance, just use your standard methodology of outputting a report to the printer:

```
* Generic call where VFP prompts the user with the
* printer dialog each time the report is run
REPORT FORM ContactListing ;
       TO PRINTER PROMPT NOCONSOLE
```

OR

```
* Generic call so user selects printer before
* report is printed, but it changes the VFP Printer
SYS(1037)
REPORT FORM ContactListing TO PRINTER NOCONSOLE
```

OR

```
* Call that has a hardcoded setting to drive the
* report to the Acrobat Printer, yet saves the
* old printer setting for reset later.
lcPDFPrinter = "Acrobat PDFWriter"
```

```
lcOldPrinter = SET("PRINTER", 2)

SET PRINTER TO NAME (lcPDFPrinter)

REPORT FORM ContactListing TO PRINTER NOCONSOLE

SET PRINTER TO NAME (lcOldPrinter)
```

Once the report is sent to the printer via the **REPORT FORM** command, the dialog shown in **Figure 2** is presented.

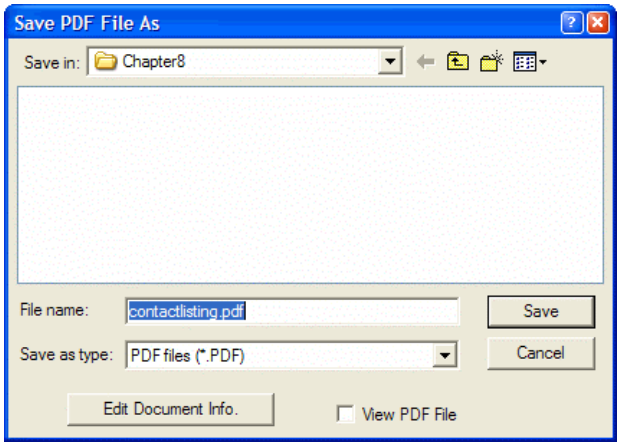


***Figure 2**. The Save PDF File As dialog allows the user to specify the name of the PDF file as well as specific document properties.*

Optionally you can hit the Edit Document Info. command button on this dialog to bring up the Acrobat PDFWriter Document Information dialog (shown in **Figure 3**).

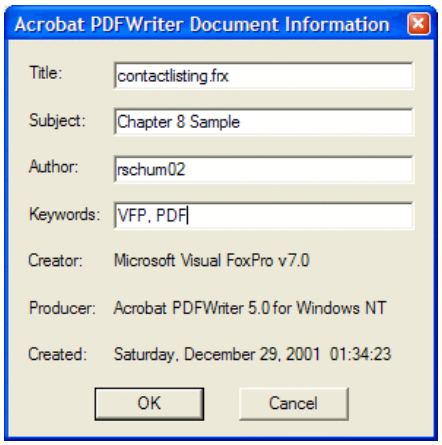


***Figure 3**. The PDF Document Information dialog provides the readers of the document key details.*

This information is stored (and can be optionally reviewed) in the PDF file that is generated (see **Figure 4**).

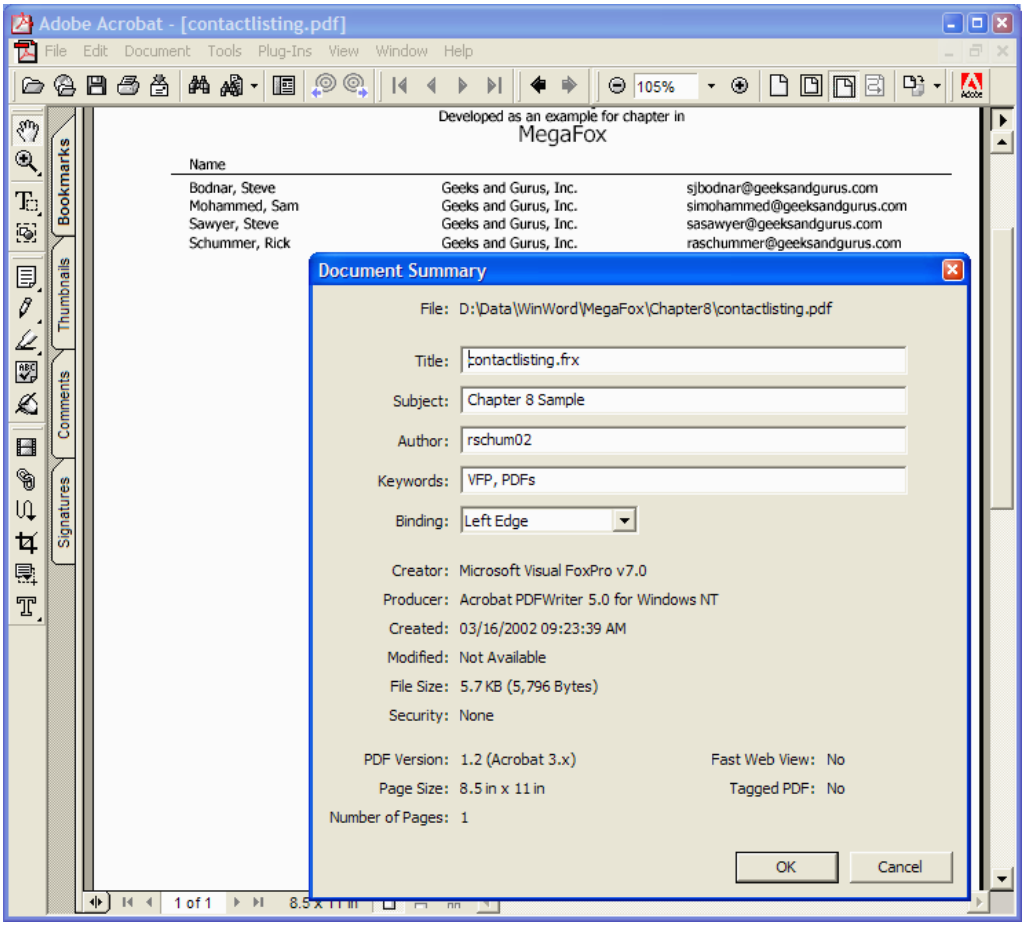


***Figure 4**. The Document Summary dialog within Acrobat will display the PDF Document information for the reader as entered by the document creator.*

The document summary information is often used by Web site search engines and indexers to make available the contents of PDF files to the people browsing their site.

What are the errors to trap when printing to PDFs? (*Example: cusAmyuniPDF::Error() of g2pdf.vcx, NoHandsAmyuniPdf.prg*)

The key to printing to PDFs (and any other printer driver selection process) is to capture the Visual FoxPro "Error loading printer driver" (error 1958). Make sure to include this trap in your error scheme or swap in a special error trap into the report printing mechanism.

```
LPARAMETERS tnError, tcMethod, tnLine

DO CASE
  CASE tnError = 1958
    THIS.lDriverError = .T.
  OTHERWISE
    AERROR(this.aErrorInfo)

    IF DODEFAULT(tnError, tcMethod, tnLine)
      MESSAGEBOX("There was a problem encountered when creating " + ;
                 "the PDF File (" + this.cPDFFileName + ")." + ;
                 CHR(13) + CHR(13) + ;
                 this.aErrorInfo[2] + " (" + ;
                 ALLTRIM(STR(this.aErrorInfo[1])) + ")", ;
                 0 + 48, _SCREEN.CAPTION)
    ENDIF
ENDCASE

RETURN
```

The biggest gotcha to watch for when printing Visual FoxPro reports to PDF is getting bitten by the hard-coded printer details. One of the better-known problems with Visual FoxPro reports is accidentally hard-coding printer driver information that gets stored in the report metadata. The information is stored in the report metadata file (FRX) in the EXPR, TAG, and TAG2 columns. If these fields have specific printer information included in the columns, Visual FoxPro will attempt to print to that printer and not the PDF driver. The symptom of this problem is having output printed on the printer when you attempt to generate a PDF file. We discussed this problem and a solution in *1001 Things You Wanted to Know About Visual FoxPro* on page 542, "How to remove printer info in production reports," and on page 496, "How to remove the printer information from Visual FoxPro reports."

How do I run PDF reports unattended using Acrobat?

(*Example: NoHandsPDF.prg*)

In a previous section we discussed the basic Visual FoxPro report print to PDF process. While this process is straightforward, it has a significant drawback in the fact that it needs an end user to interact and enter a file name before the PDF can be generated. What happens if you want to automatically generate a slew of reports from Visual FoxPro during a batch process that happens in the middle of the night? You or your clients could hire an operator who sits and watches the process and types in the file names as they are prompted, or you can head directly to the West Wind Web site and get the wwPDF50 ZIP file.

Rick Strahl has written plenty of code that allows Visual FoxPro developers to generate PDF files without the printer driver interaction prompting for a PDF file name. This class (wwPdf.prg) is available from www.west-wind.com/Webtools.asp and is available as part of the chapter source code downloadable from the Hentzenwerke Web site. The newest download available from West Wind has a change in it to better work with Acrobat 5.0. There are other classes included that work with Acrobat Distiller and the ActivePDF drivers.

The Acrobat printer driver is driven on settings available in the WIN.INI file. The wwPDF40 class manipulates the Acrobat file name settings in this INI file. The implementation of hands-free Acrobat printing is straightforward:

```
* Partial listing from NoHandsPDF.prg
SET PROCEDURE TO wwPDF ADDITIVE
SET PROCEDURE TO wwAPI ADDITIVE

loPDF = CREATEOBJECT('wwPDF40')
lcFileName   = "ContactList" + lcNow + ".pdf"
lcOutputFile = ADDBS(SYS(2023)) + lcFileName

* Use PrintReport() instead of PrintReportToString()
* IMPORTANT: FRX must have printer specified as PDFWriter
loPDF.PrintReport("ContactListing", lcOutputFile)

* Destroy the PDF Object
loPDF = .NULL.
```

Set procedure to two programs that contain all the class definitions necessary to manipulate the needed operating system INI files that contain the information used by the Acrobat PDF printer driver. Then create the PDF file without the user being prompted for a file name. The sample code is creating the PDF output in the Visual FoxPro temp directory.

You might be wondering why the sample code has a **SET REPROCESS** command. The wwPDF classes work around an issue with the PDF Writer. The printer driver is single threaded. This means that it needs to generate one report at a time. The wwPDF class sets up a table and performs a record lock until the PDF is generated. This concept of enforcing the single threaded process is called semaphore locking. If you are simultaneously printing a massive number of PDFs you might consider a different solution since this class will slow the overall throughput. Web sites that generate PDF documents on the fly might want to consider the ActivePDF since it is multi-threaded and can take advantage of multiple processors.

There is a complete whitepaper on this topic written by Rick Strahl, "Web reports with Adobe Acrobat Documents," at www.west-wind.com/presentations/pdfwriter/pdfwriter.htm. Rick Strahl also details how the PDF writing process is single threaded (important on a Web server process) and exactly how his classes work with semaphore locking to make sure that the reports are handled one by one. If your Web application is generating thousands upon thousands of Visual FoxPro reports in this manner the throughput may become an issue.

How do I run PDF reports unattended using Amyuni?

(*Example: NoHandsAmyuniPDF.prg, cusAmyuniPDF::g2pdf.vcx*)

In the previous section we demonstrated building PDF files in a hands-off mode (requiring no user interaction). This technique requires two tools, the full Acrobat version and the West Wind PDF classes. Rick Strahl is kind enough to offer his classes for free, but the Acrobat product lists for approximately $225 per license. If you are running this solution you need to buy a license for each user (or Web server) that is generating these documents. This may not sound bad for a shrink-wrapped package that costs in the tens of thousands of dollars, but what if all 50 users need this functionality? You could be adding another $10,000 to the

project implementation costs. This is where a product like the Amyuni PDF Converter comes into play.

Amyuni provides a full demonstration version of the Amyuni PDF Converter. We have included it in the chapter downloads, but a more current version might be available at the Amyuni Web site (www.Amyuni.com). The file name in the downloads is PdfSUDemoEn.exe. This needs to be installed to run the samples. The only difference between the demo and registered version is that a watermark is included on each PDF generated with the demo version.

The PDF Converter is accessed in code via an ActiveX interface or an FLL library. The examples we will demonstrate here are for the ActiveX interface. The class example (cusAmyuniPDF class in G2PDF.VCX, available in the chapter download file from Hentzenwerke.com) handles both editions so feel free to review the code for the differences between the two approaches. First you must instantiate the control and initialize it.

```
this.oPDFPrinter = CREATEOBJECT("CDINTF.CDINTF")
this.oPDFPrinter.DriverInit("PDF Compatible Printer Driver")
```

After the printer driver is initialized we need to set up the parameters to achieve the desired output. This process is handled through the custom *SetDriverParameter()* method. There are several parameters available. We have set up several properties in the cusAmyuniPDF custom class to handle the options. The method code is as follows:

```
* cusAmyuniPDF.SetDriverParameter() method
* Do not prompt for file name
#DEFINE ccPDF_NOPROMPT            1
* Use file name set by SetDefaultFileName
* else use document name
#DEFINE ccPDF_USEFILENAME         2
* Concatenate files, do not overwrite
#DEFINE ccPDF_CONCATENATE         4
* Disable page content compression
#DEFINE ccPDF_DISABLECOMPRESSION  8
* Embed fonts used in the input document
#DEFINE ccPDF_EMBEDFONTS          16
* Enable broadcasting of PDF events
#DEFINE ccPDF_BROADCASTMESSAGES   32

IF NOT ISNULL(this.oPDFPrinter)
  * Set the destination file name.
  this.oPDFPrinter.DefaultFileName = this.cPDFFileName

  * Set resolution to to the desired quality
  this.oPDFPrinter.Resolution      = this.nResolution

  * Update driver info with resolution information
  this.oPDFPrinter.SetDefaultConfig()

  * Note: Message broadcasting should be enabled
  * in order to insert bookmarks from VFP.
  * But see the notes in the SetBookmark method
```

```
  this.oPDFPrinter.FileNameOptions = ;
    IIF(this.lPrompt, 0, ccPDF_NOPROMPT + ccPDF_USEFILENAME) + ;
    IIF(this.lBookmarks, ccPDF_BROADCASTMESSAGES, 0) + ;
    IIF(this.lConcatenate, ccPDF_CONCATENATE, 0) + ;
    IIF(this.lCompression, 0, ccPDF_DISABLECOMPRESSION) + ;
    IIF(this.lEmbedFonts, ccPDF_EMBEDFONT, 0)

  * Save the current Windows default printer
  * so we can restore it later.
  this.oPDFPrinter.SetDefaultPrinter()
ELSE
  * Handle settings via the FLL.
ENDIF

RETURN
```

Now the driver is ready to produce the PDF file. At this point you have made settings to have the user not be prompted for a file name (default in this example), and to indicate whether bookmarks are generated (FLL option only), if the contents are concatenated with previous output, if the PDF is compressed (a default for PDFs), and if fonts are embedded. This is not that much work. The Visual FoxPro report can now be generated with the following code:

```
* Set the VFP printer name to the PDF printer, and print the report.
this.cOldPrinterName = SET("printer", 2)
SET PRINTER TO NAME (THIS.cAmyuniDriver)
REPORT FORM (this.cReportName) NOEJECT NOCONSOLE TO PRINTER
```

The class also handles the resetting of the original printer driver and cleans up the object references in the Destroy method of the object. Modifications or enhancements to this class could also forward a text file or HTML output generated from your applications to a PDF file as well. Amyuni has other drivers available to support creation of HTML and text (via the Rich Text Format).

If you are using the Amyuni FLL interface you will need the FLLINTF.FLL file provided by Amyuni. This file is installed in the same directory as the Amyuni ActiveX controls and sample files. Even if you are not using the FLL interface you will need to include this directory in the Visual FoxPro path to recompile the class since the code is included for this option and the FLL is referenced.

How do I email a Visual FoxPro report? (*Example: MailPDFBatch.prg*)

One question that gets asked frequently on the support forums is: How can I e-mail the results of a report? One approach is to run the Visual FoxPro report to a PDF file and have the application attach it to an e-mail. There are a number of e-mail components available that integrate with Visual FoxPro. It is beyond the scope of this chapter to get into the nuts and bolts of automating a MAPI compliant e-mail client, but we wanted to reveal one of the most useful implementations of Acrobat PDFs in our applications. There are numerous examples of integrating e-mail with Visual FoxPro in Chapter 4, “Sending and Receiving E-mail.” This example will leverage another class from West Wind called wwIPStuff (see **Listing 1**).

***Listing 1**. A program that uses the wwIPStuff class and DLL from West Wind to e-mail a Visual FoxPro report as a PDF file.*

```
LPARAMETERS tlEmail

#INCLUDE foxpro.h

SET EXCLUSIVE OFF
SET DELETED ON
SET PROCEDURE TO wwPDF ADDITIVE
SET PROCEDURE TO wwAPI ADDITIVE
SET PROCEDURE TO wwUtils ADDITIVE
SET PROCEDURE TO wwEval ADDITIVE
SET CLASSLIB  TO wwIPStuff ADDITIVE

OPEN DATABASE pdfsample
SET DATABASE TO pdfsample

IF NOT USED("curMailing")
  USE pdfsample!v_geekscontactlist IN 0 AGAIN ALIAS curMailing
ELSE
  REQUERY("curMailing")
ENDIF

IF NOT USED("curList")
  USE pdfsample!v_geekscontactlist IN 0 AGAIN ALIAS curList
ELSE
  REQUERY("curMailing")
ENDIF

IF NOT USED("EmailInfo")
  USE pdfsample!EmailInfo IN 0 AGAIN ALIAS EmailInfo
ENDIF

IF NOT USED("EmailHistory")
  USE pdfsample!EmailHistory IN 0 AGAIN ALIAS EmailHistory
ENDIF

loIPMail = CREATEOBJECT('wwIPStuff')
loPDF    = CREATEOBJECT('wwPDF40')

SELECT curMailing

SCAN
  lcFileName   = ALLTRIM(curMailing.First_Name) + ;
                 ALLTRIM(curMailing.Last_Name) + ;
                 ALLTRIM(STR(curMailing.Contact_Id)) + ".pdf"
  lcOutputFile = ADDBS(SYS(2023)) + lcFileName

  * Generate the PDF file
  SELECT curList
  loPDF.PrintReport("ContactListing", lcOutputFile)

  loIPMail.cMailServer  = ALLTRIM(emailinfo.cMailServe)
  loIPMail.cSenderEmail = ALLTRIM(emailinfo.cSender)
  loIPMail.cSenderName  = ALLTRIM(emailinfo.cSenderName)

  loIPMail.cRecipient   = ALLTRIM(curMailing.Email_Name)
  loIPMail.cSubject     = ALLTRIM(emailinfo.cSubject)
```

```
  loIPMail.cMessage     = ALLTRIM(emailinfo.cMessage) + ;
                          ALLTRIM(emailinfo.cSignature)

  * Here is where we attach the PDF file
  IF FILE(lcOutputFile)
    loIPMail.cAttachment  = lcOutputFile
  ENDIF

  lcSentMsg = "To: " + loIPMail.cRecipient + ;
              CHR(13) + "From: " + loIPMail.cSenderEmail + ;
              IIF(EMPTY(loIPMail.cCCList), SPACE(0), CHR(13) + "CC: " + ;
                                          loIPMail.cCCList) + ;
              IIF(EMPTY(loIPMail.cBCCList), SPACE(0), CHR(13) + "BCC: " + ;
                                          loIPMail.cBCCList) + ;
              CHR(13) + "Subject: " + loIPMail.cSubject + ;
              CHR(13) + loIPMail.cMessage

  * Only send the list of produced
  IF FILE(lcOutputFile)
    * Send only if passing parameter, allows testing
    * without sending the email
    IF tlEmail
      llResult = loIPMail.SendMail()
    ELSE
      llResult = .F.
    ENDIF
  ELSE
    llResult = .F.
  ENDIF

  IF !llResult
    WAIT WINDOW "No email message to " + loIPMail.cRecipient + " (" + ;
                loIPMail.cErrorMsg + ")" NOWAIT
    lcSentMsg = lcSentMsg + CHR(13) + CHR(13) + ;
                IIF(tlEmail, "Intended to email", "Not intended to email") +;
                CHR(13) + ;
                "ERROR: " + loIPMail.cErrorMsg

    INSERT INTO emailhistory (tTimeStamp, lSentEmail, mMessage, cRecipient) ;
      VALUES (DATETIME(), .F., lcSentMsg, curMailing.Email_Name)
  ELSE
    WAIT WINDOW "Sent message to " + loIPMail.cRecipient NOWAIT
    lcSentMsg = lcSentMsg + CHR(13) + CHR(13) + "Message sent successfully"

    INSERT INTO emailhistory (tTimeStamp, lSentEmail, mMessage, cRecipient) ;
      VALUES (DATETIME(), .T., lcSentMsg, curMailing.Email_Name)
  ENDIF
ENDSCAN

loPDF    = .NULL.
loIPMail = .NULL.

USE IN (SELECT("curMailing"))
USE IN (SELECT("curList"))
USE IN (SELECT("emailhistory"))
USE IN (SELECT("emailinfo"))
USE IN (SELECT("contacts"))

RETURN
```

The example code list is only a partial list of the code in the example program. The wwIPStuff included in the chapter downloads is a shareware version that is available on the West Wind Web site (www.west-wind.com). It demonstrates the simple implementation of the wwIPStuff class and corresponding DLL file, which are included in Web Connect, or can be purchased separately. The shareware version will display a WAIT WINDOW, but allows complete concept/prototype testing before purchasing the commercial product.

The basic idea is to generate the PDF file and attach it to an e-mail. Since this implementation directly sends the e-mail via Simple Mail Transfer Protocol (SMTP), it bypasses all e-mail clients. This means that there will be no audit trail of the sent mail item in a Sent Item folder. While it is nice to trust that the e-mail is safely transferred via the Internet, our customers like to have a record that the e-mail was sent and some details about what was included. The second half of the program provides a basic audit trail of the e-mail, if it was sent successfully, and if not, what error occurred.

To test this program out you will need to change a few columns in the EmailInfo table. The cMailServer is the SMTP server for your e-mail account, cSender is your e-mail address, cSenderName is your name, cMessage is the narrative contents of the message in the e-mail, and cSignature allows for an optional signature line for the message.

We set up the program with a parameter (tlEmail) so the program can be run without actually sending the e-mail. If you run this program with the parameter set to `.T.`, please change the e-mail addresses in the Contacts table to something you will receive and not the chapter author and his partners.

How can I replace the Visual FoxPro Report print preview? (*Example: AltPreview.scx*)

If you poll Visual FoxPro developers and have them note one weakness in Visual FoxPro, our guess is that a big percentage of them would point to the Report Designer Preview mode. It has not had a major enhancement since the days of version 2.x. There are plenty of issues with the display depending on the printer drivers, video drivers, and monitor resolution. The zoom feature has limited percentage settings. It has no drill down capability and shows its age by not displaying hyperlinks. One day we thought, why not use Acrobat to act as the report print preview instead of the standard Visual FoxPro method?

Previously in this chapter we demonstrated a method to generate the PDF file without user interaction. Now all we need is a method of displaying the document in the Acrobat Reader. Not a problem, the following line of code works just fine on our PC:

```
RUN /n1 ;
    "C:\Program Files\Adobe\Acrobat 5.0\Acrobat\Acrobat.exe" ;
    "C:\My Documents\MemberList200008.PDF"
```

So now we need a way to make the call generic. There are several solutions to this. We can store the location in a configuration table or INI file. While this works it is just one more thing that the users need to maintain and can possibly set up incorrectly, which potentially will lead to another support call. So how can you determine the location of Acrobat? Fortunately,

Acrobat registers itself in the Windows Registry and the executable is stored in several keys. The key that seems appropriate for this exercise is:

```
[HKEY_CLASSES_ROOT\AcroExch.Document\shell\print\command]
```

The results will differ based on which version of Acrobat is installed, full product or just the Reader, and the OS platform you are using. It is important to note that you will need the full product to generate the PDF files to start with unless you have a product like the Amyuni PDF Converter. On our computers the Registry entry consists of the following values:

Acrobat (full):
C:\Program Files\Adobe\Acrobat 5.0\Acrobat\Acrobat.exe

Acrobat Reader:
C:\Program Files\Adobe\Acrobat 5.0\Reader\AcroRd32.exe

So with this functionality we can now use a Registry class to grab the location of the executable. The example created (ALTRPTPREVIEW.SCX::*RptPreview()* method) will use the same technique as the Acrobat hands-free example (including the wwPDF50 classes from Rick Strahl). It uses the Registry class that comes as part of the Fox Foundation Classes (FFC) to determine the location of Acrobat and executes the Reader with the PDF file as the parameter.

```
lcRegFile = HOME(2)+"classes\registry.prg"
lcAppKey  = ""
lcAppName = ""
loPDF     = CREATEOBJECT('wwPDF40')

* Check for the existence of the registry class
IF NOT FILE(lcRegFile)
  MESSAGEBOX("Registry class was not found (" + lcRegFile + ")")
  RETURN
ENDIF

* Instance the Registry object
loReg     = NEWOBJECT("FileReg", lcRegFile)

* Get Application path and executable
lnErrNum  = loReg.GetAppPath("PDF", @lcAppKey, @lcAppName)

IF lnErrNum != 0
  MESSAGEBOX("No information available for Acrobat application.")
  RETURN
ENDIF

* Remove switches here (i.e., C:\EXCEL\EXCEL.EXE /e)
IF ATC(".EXE", lcAppName) # 0
  lcAppName = ALLTRIM(SUBSTR(lcAppName, 1, ATC(".EXE", lcAppName) + 3))

  IF ASC(LEFT(lcAppName, 1)) = 34    && check for long file name in quotes
    lcAppName = SUBSTR(lcAppName, 2)
  ENDIF
ENDIF
```

Now that you have the location of the Acrobat executable you can proceed with the building of the file and shell out to Acrobat in "preview mode."

```
* Build the file name for the PDF
lcFileName   = "ContactList" + lcNow + ".pdf"
lcOutputFile = ADDBS(SYS(2023)) + lcFileName

* Generate the PDF file
loPDF.PrintReport("ContactListing", lcOutputFile)

* Run Acrobat or Acrobat Reader
RUN /n1 ;
    &lcAppName ;
    &lcOutputFile
```

The RUN command does not wait for the Acrobat application to be shut down. This is important in the fact that any code that follows the preview will execute. Therefore, do not run code to clean up the PDF files because they are open.

It should be noted that repeated calls to run any version of Acrobat will open up another PDF file in the one single instance of Acrobat. This has no effects on the ability for the user to review any of the files. As with anything in the computing world, the limits are memory, file handles, and other system resources.

Another way to do this is:

```
* Example call:
DO shell WITH "ContactListing.PDF", ;
              "C:\My Documents\", ;
              "open"

* Program  : Shell.prg
*   WinApi : ShellExecute
*  Function: Opens a file in the application
*            that it's associated with.
*      Pass: lcFileName -  Name of the file to open
*
*    Return:  2 - Bad Association (ie, invalid URL)
*            31 - No application association
*            29 - Failure to load application
*            30 - Application is busy
*
*            Values over 32 indicate success
*            and return an instance handle for
*            the application started (the browser)
LPARAMETERS tcFileName, tcWorkDir, tcOperation

LOCAL lcFileName, ;
      lcWorkDir, ;
      lcOperation
```

```
IF EMPTY(tcFileName)
  RETURN -1
ENDIF

lcFileName  = ALLTRIM(tcFileName)
lcWorkDir   = IIF(TYPE("tcWorkDir") = "C", ;
                  ALLTRIM(tcWorkDir),"")
lcOperation = IIF(TYPE("tcOperation")="C" AND ;
                  NOT  EMPTY(tcOperation), ;
                  ALLTRIM(tcOperation),"Open")

* ShellExecute(hwnd, lpszOp, lpszFile, lpszParams,;
*              lpszDir, wShowCmd)
*
* HWND hwnd         - handle of parent window
* LPCTSTR lpszOp    - address of string for operation to perform
* LPCTSTR lpszFile  - address of string for filename
* LPTSTR lpszParams - address of string for executable-file parameters
* LPCTSTR lpszDir   - address of string for default directory
* INT wShowCmd      - whether file is shown when opened
DECLARE INTEGER ShellExecute ;
        IN SHELL32.DLL ;
        INTEGER nWinHandle,;
        STRING cOperation,;
        STRING cFileName,;
        STRING cParameters,;
        STRING cDirectory,;
        INTEGER nShowWindow

RETURN ShellExecute(0,lcOperation,lcFilename, SPACE(0), lcWorkDir,1)
```

So what are some of the advantages of this reporting alternative? In our opinion, it addresses some of the Visual FoxPro Report Writer drawbacks. It mainly addresses the weakness of the preview zoom (or as it is really known as, "lack of zoom"). The Acrobat Reader provides super zoom capability (12.5% up to 1600%). Other nice-to-have features are having multiple pages visible at one time with continuous mode, a search feature, and a true What-You-See-Is-What-You-Get (WYSIWYG). You can also view multiple PDF reports since the Acrobat Reader can open multiple PDF files.

Visual FoxPro developers have been challenged by the Visual FoxPro Report Designer and have not been bashful about voicing these issues. Microsoft has repeatedly noted that there will be little to nothing addressed with the existing Report Designer in future versions of Visual FoxPro. Microsoft has also noted that we live in a component world. This is a beautiful example of that component world reaping benefits for our clients. The example uses Rick Strahl's wwPDF class to avoid the user interaction when the PDF file is generated before it is previewed in Acrobat. The code can be altered to use any one of the other PDF generators that are available to developers.

How do I present Acrobat PDFs in a Visual FoxPro form?

(*Example: PdfDisplay5.scx, PdfDisplay5a.scx*)

If you have Acrobat or the Acrobat Reader product you will also have the ActiveX control that will display a PDF file in a Visual FoxPro form. There are two controls that appear in the Tools | Options dialog on the Controls page. The control you want to work with is Acrobat

control for ActiveX. The other control, Adobe Acrobat document, only allows you to hard-code the PDF file that is displayed.

We want to give you a word of caution before moving into development with this control. We originally developed the samples with the control included in Acrobat 4.0. These samples have worked flawlessly. In March 2001 Acrobat 5.0 version was release. We have crashed Visual FoxPro 6 and Visual FoxPro 7 a number of times with the newest version. The examples presented have worked around the C5 errors. Adobe states specifically on its Web site that this control was designed specifically to work with Microsoft's Internet Explorer, yet discusses its use with developer tools like Visual Basic. So tread carefully with the examples and implementation in applications.

Like all ActiveX controls, first you will need to select the Acrobat control for ActiveX in the Controls tab of the Visual FoxPro Options dialog (see **Figure 5**).

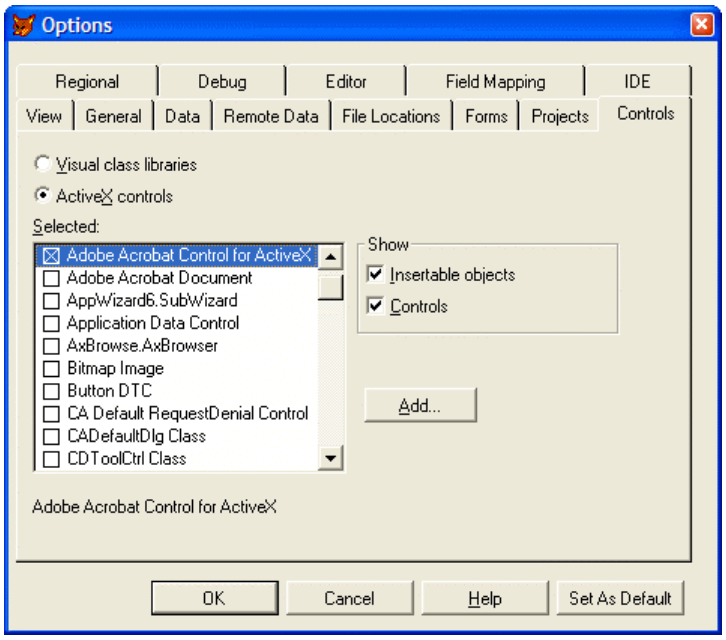


***Figure 5**. The Acrobat control for ActiveX is available in the Controls tab of the Visual FoxPro Options dialog.*

Building the form is straightforward. Drop the control from the ActiveX palette on the Visual FoxPro Form Controls toolbar to a Visual FoxPro form (see **Figure 6**).

***Figure 6**. The Acrobat control is the middle toolbar button (with the Acrobat symbol).*

The property that needs to be set and/or bound to a Visual FoxPro control is *SRC*. This tells the Acrobat control which PDF file to load and display. The *SRC* property can be set dynamically, which reloads the selected PDF file in the viewer (this worked fine in Acrobat 4 and causes OLE errors in Acrobat 5 unless set in the form Init method). The example form (PDFDISPLAY5.SCX, included in the downloads available from www.Hentzenwerke.com) takes a parameter, which is the PDF file name, and sets the *SRC* property of the PDF ActiveX control.

```
* PdfDisplay5.scx Init()
LPARAMETERS tcPdfFileName

this.Resize()

IF VARTYPE(tcPdfFileName) = "C" AND FILE(FULLPATH(tcPdfFileName))
  this.olePDF.SRC = FULLPATH(tcPdfFileName)
ELSE
  this.olePDF.SRC = FULLPATH(this.olePDF.SRC)
ENDIF

this.olePDF.setFocus()
this.olePDF.setZoom(150)

RETURN
```

The sample form has a couple of things you should note before trying to run it. The first is that you must have the ActiveX control registered on your PC. The second is that we have hard-coded the PDF file name in the *SRC* property. There is a good chance that your directory structure does not match ours, so some changes will need to be implemented before running the form or you will need to pass in the parameter, which is the PDF file name (fully pathed or available on the Visual FoxPro path). If the PDF is not available, the form is displayed empty since Acrobat cannot load the PDF.

The form is displayed (see **Figure 7**) with the PDF visible. Do not be surprised by the Acrobat splash screen. This is displayed when the Acrobat ActiveX control is instanced (the same behavior is displayed when a PDF file is opened in Internet Explorer). All of the toolbars that are included in the Acrobat Reader (or the full version if this is what is loaded on the PC) are available in your Visual FoxPro form, including tools to zoom in and out, print the document, search for text, change pages, and save it off to another file. Even items like Bookmarks and Thumbnails are available in the ActiveX control.

There are a number of methods that can be called to change the behavior of the PDF viewer. Unfortunately, there is no documentation in the ActiveX control properties dialog that describes the method parameters, nor is there an associate Help file. We can open up the ActiveX control (PDF.OCX) or the control's typelib file (PDF.TLB) to see what the parameters are. Still, there is no specific documentation that we could find before assembling this chapter. In Visual FoxPro 6 you need to use the Class Browser (see **Figure 8**); in Visual FoxPro 7 you will need to use the new Object Browser (see **Figure 9**).

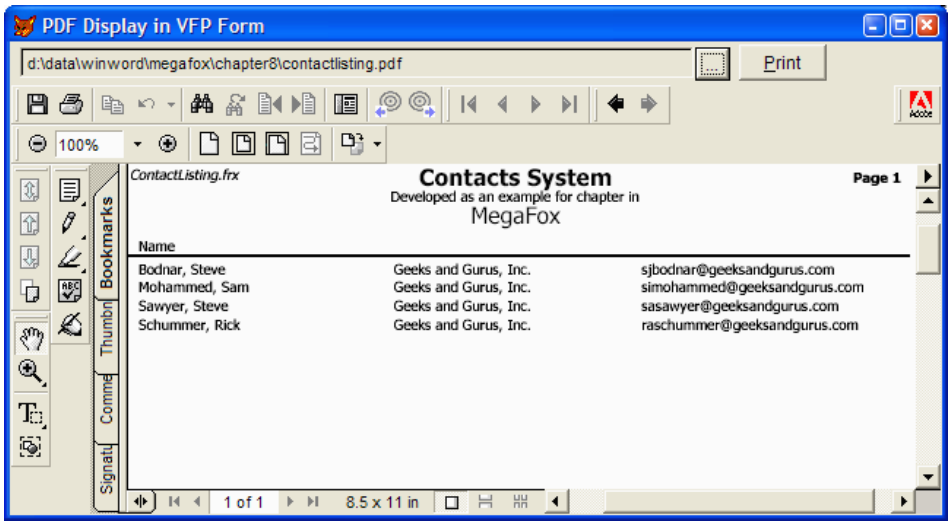


***Figure 7**. This is a PDF file displayed in a Visual FoxPro form. The Print command button will display the printer selection dialog for the user.*

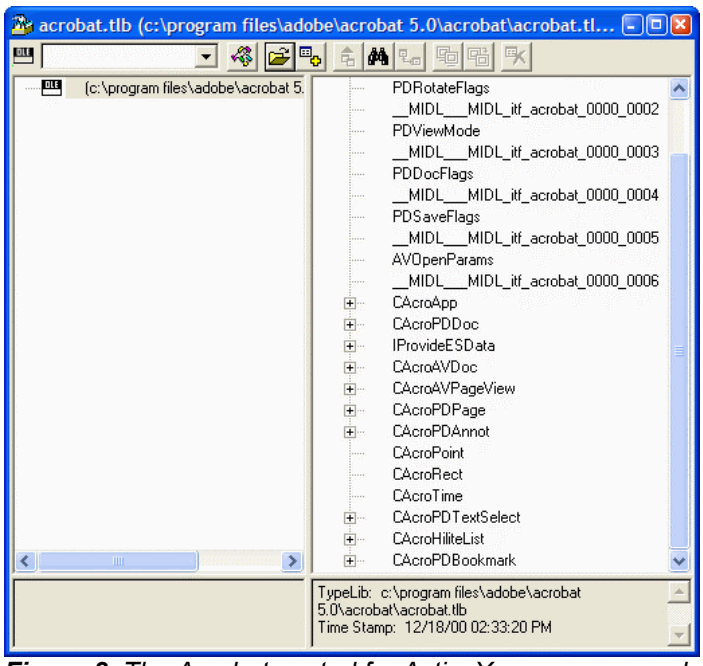


***Figure 8**. The Acrobat control for ActiveX exposes a number of methods for the developer to interact with the control in the Visual FoxPro form. This shows the features exposed in the Visual FoxPro 6 Class Browser.*

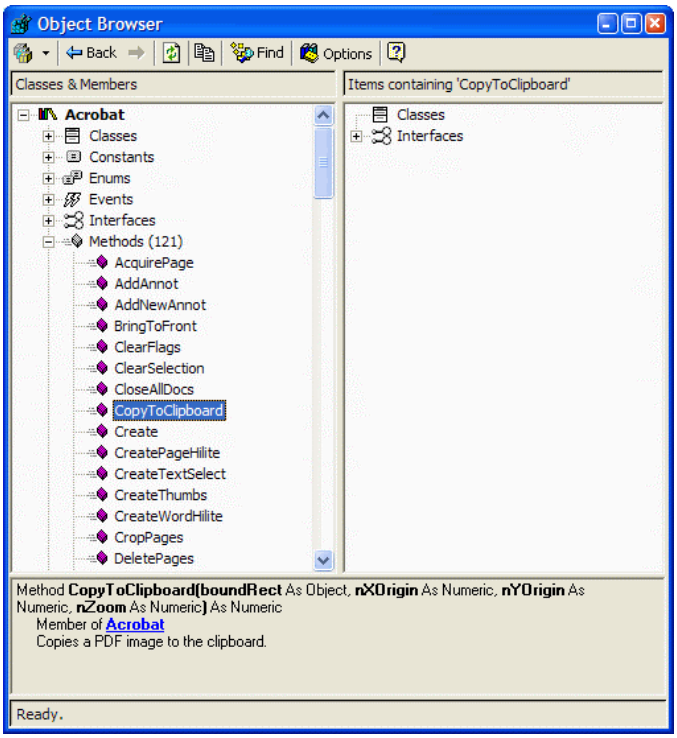


***Figure 9**. To view the property, events, and methods in Visual FoxPro 7 you need to use the Object Browser.*

All of the features you use in Acrobat Reader via the menus and toolbars are exposed in the Acrobat ActiveX control. There may be methods that might be handy to execute via your own exposed interface. A number of the Reader features are exposed through an interface of properties and methods. Note the method names usually start out with a lowercase name (visible in the Object Browser and the Acrobat JavaScript documentation). This is due to the standard that JavaScript uses, which is the native "macro" language included in Acrobat.

The *printWithDialog()* is nice because it automatically displays the printer selection and print driver option dialog that Acrobat displays when you select the File | Print menu option. You can also print directly to the Windows default printer with the *Print()* method. There are a number of print methods to suit most tastes. The *gotoLastPage()* method could be used in the cases when the customer likes to view the grand total information on the report, which is on the last page, before reviewing the details. If your users prefer to see the report zoomed at a specific percentage you can use the *setZoom()* method.

There is a second method to displaying PDFs in a Visual FoxPro form. If you have the full Acrobat product you will also have the ActiveX interface that will display a PDF file in a VFP form. This interface is *not* loaded with the Reader edition of Acrobat. However, this object is well documented, both in the type library and in the Acrobat Software Developers Kit (SDK).

The Acrobat Software Developers Kit can be downloaded from the Adobe developer page located at http://partners.adobe.com/asn/developer/acrosdk/acrobat.html.

The easiest way to work with this technique is to use the new VFP 7 Object Browser. Open up the Acrobat 5.0 Type Library object (see **Figure 10**). The main object is located under Interfaces. It is called CAcroAVDoc. This interface has the capability to get at the other needed interfaces as well as display the PDF. This object is created in the *Init* of the form.

```
* Form Init()
LPARAMETERS tcPDF

IF DODEFAULT(tcPDF)
  this.oAVDoc = CREATEOBJECT("AcroExch.AVDoc")
  this.Navigate(tcPDF)
ENDIF

RETURN
```

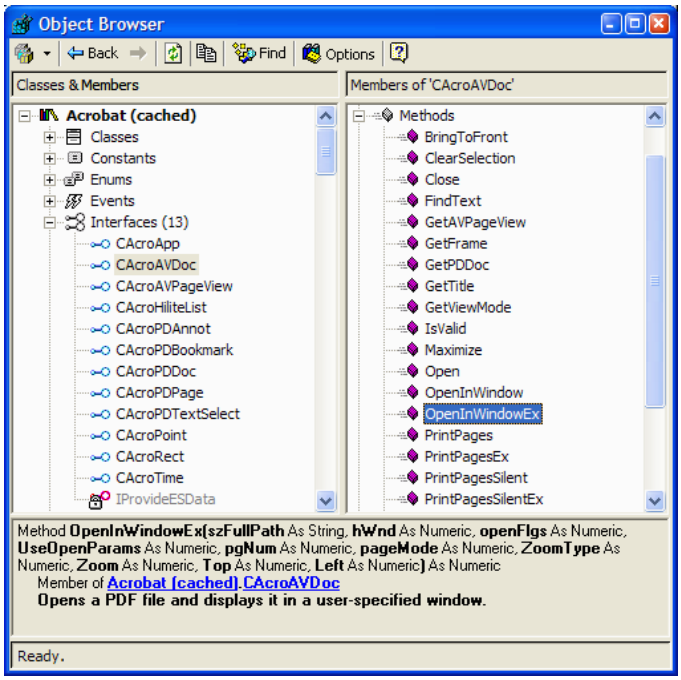


***Figure 10**. The Acrobat 5.0 Type Library object has documented interfaces, methods, and constants.*

The form will optionally accept a PDF file name as a parameter. If the Acrobat object cannot be instantiated the *Error()* method will trap the condition and disable the user interface objects on the form. After the object is created, the custom *Navigate()* method is called to open

up and display the PDF file. To open the PDF file and display it in the form we use the new Visual FoxPro *Hwnd* property as a parameter to the *OpenInWindowEx()* method. This allows Acrobat to display itself in a Visual FoxPro form.

```
* Form Navigate()
LPARAMETERS tcPDF

* Constants extracted from Acrobat 5.0 Type Library via Object Browser

#DEFINE AVZoomNoVary            0      && Fixed value zoom.
#DEFINE AVZoomFitPage           1      && Fit page to window.
#DEFINE AVZoomFitWidth          2      && Fit page width to window.
#DEFINE AVZoomFitHeight         3      && Fit page height to window.
#DEFINE AVZoomFitVisibleWidth   4      && Fit visible width to window.
#DEFINE AVZoomPreferred         5      && Use page's preferred zoom.
#DEFINE pdRotate0               0      && Rotated 0 degrees.
#DEFINE pdRotate90             90      && Rotated 90 degrees.
#DEFINE pdRotate180           180      && Rotated 180 degrees.
#DEFINE pdRotate270           270      && Rotated 270 degrees.
#DEFINE PDDontCare              0      && Leave the view mode as it is.
#DEFINE PDUseNone               1      && Display the document without
                                       && bookmarks or thumbnails.
#DEFINE PDUseThumbs             2      && Display the document and thumbnail
                                       && images.
#DEFINE PDUseBookmarks          3      && Display the document and bookmarks.
#DEFINE PDFullScreen            4      && Display the document in full screen
                                       && mode.
#DEFINE PDDocNeedsSave          1      && Document has been modified and needs
                                       && to be saved.
#DEFINE PDDocRequiresFullSave   2      && Document cannot be saved
                                       && incrementally it must be written
                                       && using PDSaveFull.
#DEFINE PDDocIsModified         4      && Document has been modified.
#DEFINE PDDocDeleteOnClose      8      && Document is based on a temporary
                                       && file.
#DEFINE PDDocWasRepaired       16      && Document was repaired when it was
                                       && opened.
#DEFINE PDDocNewMajorVersion   32      && Document's major version is newer
                                       && than current.
#DEFINE PDDocNewMinorVersion   64      && Document's minor version is newer
                                       && than current.
#DEFINE PDDocOldVersion       128      && Document's version is older than
                                       && current.
#DEFINE PDDocSuppressErrors   256      && Don't display errors.
#DEFINE PDDocIsEmbedded       512      && Document is embedded in a compound
                                       && document.
#DEFINE PDDocIsLinearized    1024      && Document is linearized (get only).
#DEFINE PDDocIsOptimized     2048      && Document is optimized.
#DEFINE PDSaveIncremental       0      && Write changes only.
#DEFINE PDSaveFull              1      && Write the entire file.
#DEFINE PDSaveCopy              2      && Write a copy of the file into the
                                       && file.
#DEFINE PDSaveLinearized        4      && Save the file in a linearized
                                       && fashion.
#DEFINE PDSaveWithPSHeader      8      && Writes a PostScript header as part of
                                       && the saved file.
#DEFINE PDSaveBinaryOK         16      && Specifies that it's OK to store in
                                       && binary file.
```

```
#DEFINE PDSaveCollectGarbage   32      && Remove unreferenced objects, often
                                       && reducing file size.
#DEFINE AV_EXTERNAL_VIEW        1      && Open the document with the tool bar
                                       && visible.
#DEFINE AV_DOC_VIEW             2      && Draw the page pane and scrollbars.
#DEFINE AV_PAGE_VIEW            4      && Draw only the page pane.

IF VARTYPE(tcPDF) = "C"
  IF FILE(tcPDF)
    WITH this.oAVDoc
      * It's important to close each doc, every time. If you don't, when you
      * try viewing the same page, it won't display anything - you have to kill
      * the object references and close VFP, plus kill Adobe. It maintains a
      * collection of open documents , but we are only using one document at a
      * and the zero makes sure the document is not saved. It keeps things
      * simple, and to keep the memory usage to a minimum.
      .Close(0)

      .OpenInWindowEx(tcPDF, this.Hwnd, AV_EXTERNAL_VIEW, ;
                      .T., 0, PDUseNone, AVZoomPreferred, ;
                      100 , 30, 0)

      this.oAVPage  = .GetAVPageView()

      * Set the zoom options
      this.ResizeAcrobat()

      IF !ISNULL(this.oAVPage)
        * Turn on, preset the zoom control on the form. Then zoom to the
        * correct PDF size.
        this.oAVPage.ZoomTo(0, 100)
        this.oAvPDDoc = .GetPDDoc()
      ENDIF

      this.cOpenPDF = this.FormatFileName(tcPDF)
      this.Refresh()
    ENDWITH
  ELSE
    MESSAGEBOX("PDF File selected does not exist", ;
               0 + 64, this.Caption)
  ENDIF
ENDIF

RETURN
```

We decided to include all the **#DEFINE**s so you can see the various options available. The *Navigate()* method first closes an existing PDF if one is open, and then opens up the selected PDF and displays it. The *Navigate()* method also instantiates two more Acrobat objects. The first is based on the CAcroAVPageView interface. There are a number of methods available on this object to manipulate to a specific location in the document and determine what the user will see. Methods include *ScrollTo()* (to scroll to a specific location on a page), *ZoomTo()* (to zoom the document to a certain percentage), *DoGoBack()* (to return to the previous position in the view history stack), and *DoGoForward()* (to return to the next view in the history stack). The second is based on the CAcroPDDoc interface. This object provides methods *GetNumPages()* (to find out the number of pages, handy when printing the documents or ranges of pages), *GetFileName()* (to know what PDF is open), *DeletePages()*/*CreateThumbs()*/

DeleteThumbs() (if you want to manipulate the contents of the documents), and *Save()* (does what you would expect). We did not implement all of these methods, as we thought that it would take all the fun away from you, and so we've left that as an exercise for you to become familiar with the different objects.

The sample form will open up without showing a PDF if you do not pass the file as a parameter. The user can then use the ellipsis button (three dots) to select a PDF. This button uses the `GETFILE()` function to obtain the PDF name, and then calls the *Navigate()* method. The user can resize it and have the PDF viewer resize itself as well. The only real drawback of this technique is that it is only available to users that have the full version of Acrobat installed.

What is Acrobat Forms Author technology? (*Example: SHAppBuildPermitData.pdf*)

Acrobat ships with a cool feature called Acrobat Forms Author Technology that is provided via a plug-in (add-on or extension to the base product). This technology allows end users to convert paper forms into electronic forms that have the exact look of the original paper forms. These forms can be displayed in Acrobat and the end users can enter data in the same exact format they used when filling out the paper directly. This form might be a company standard, an industry directive, or a governmental dictate.

If your users are as demanding as ours, you have probably run into the situation where you have been asked to produce an interface form that duplicates the existing paper version. You go off to develop this slick interface and demo the prototype to the users. The first thing they mention is that it does not mimic the paper version of the form "exactly." The flip side is printing reports that mimic the paper version. While this is usually easier than the data entry part of the equation, generating reports with various lines and boxes, detail lines that exceed the facilities of the Visual FoxPro Report Designer or even some of the third-party report writers can be a challenge. Once in a while it is impossible. Acrobat Forms can assist us in getting data via data entry and outputting data to the forms for printing (see **Figure 11**).

The Forms Author plug-in capability is included with the full version of Acrobat. The data entry mode is available in the Reader version as well as full Acrobat, and a new product called Acrobat Approval. So what are the advantages? For one, the forms can be replicated electronically just like they are on paper. Since Acrobat printing is truly What-You-See-Is-What-You-Get (WYSIWYG), the forms can be printed after being filled in. They can be saved with the data entered, which provides an audit trail. Most importantly, the information can be extracted and saved in a database for further analysis.

Visual FoxPro developers might be asking the question, why would I need Acrobat Forms when I have a great forms designer in Visual FoxPro? The difference is that Acrobat Forms can also be implemented in a distributed environment via the Internet without the overhead of the ActiveDoc technology used in Visual FoxPro. This means that the PDF file can be accessed on the Web, users can enter in data, and the information can be submitted to the Web server for processing and the data extracted and stored into a database.

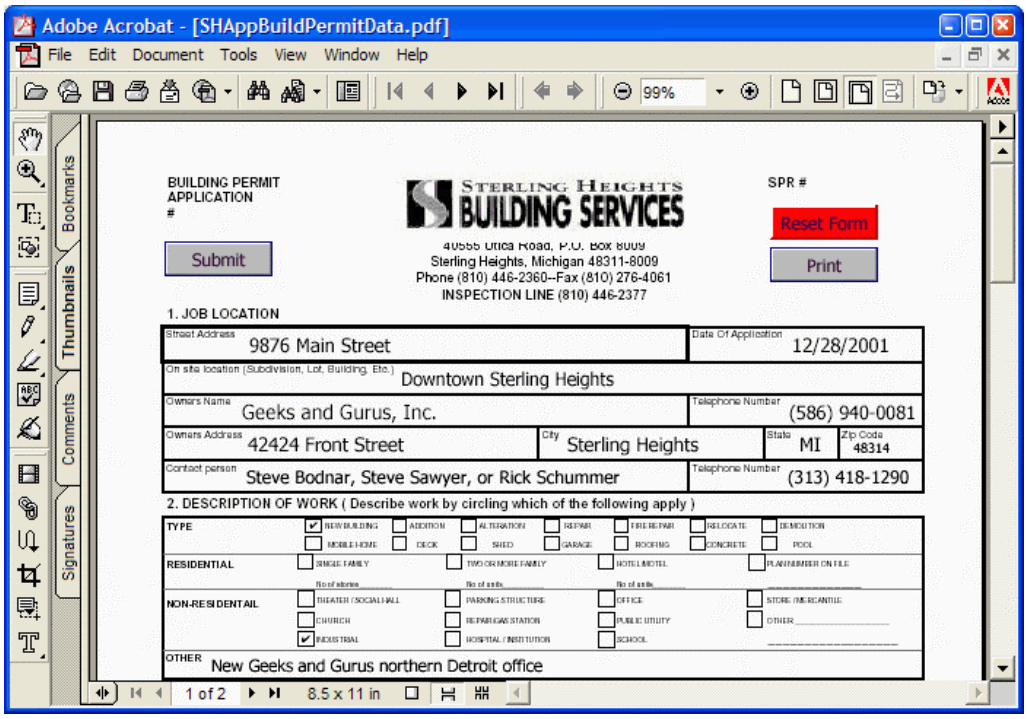


***Figure 11**. This form is the city of Sterling Heights Building Permit form with some data filled in as the user would see it in Acrobat.*

There are a couple of concepts in developing these forms that are very familiar to Visual FoxPro developers. The Acrobat Forms "designer" has similar functionality as the Visual FoxPro form designer. You change the mode of Acrobat with the PDF from "entry" to "designer" via the Form Tool icon on the left-side toolbar icon (second from the bottom on the left side toolbar in **Figure 12**). This toggles the mode so the form editor is available. Right-clicking on any object will bring up the shortcut menu. One of the many menu options is Properties. Selecting this option will introduce the Acrobat form field property sheet (see **Figure 13** for one page in this dialog). There are properties to name the objects, comment their use, adjust fonts, format the entry, set colors, require data, make it read only, have default values, set the tab order, and align the text. There are settings to run code for events and perform validation. Sound familiar? Sounds like what we do with the Visual FoxPro Class and Form Designers on a regular basis. Object types include Text (TextBox), CheckBox, ComboBox, ListBox, RadioButton (OptionGroup), Button (CommandButton) and Signature (no Visual FoxPro equivalent).

The property dialog is very comfortable to Visual FoxPro developers. The biggest difference is that the code is written in JavaScript. Dropping objects on the PDF form is completed by changing the PDF into "designer mode" as noted earlier, and clicking and dragging to size the new object. This will open the field properties dialog. You select the object type and start setting the various properties. Each subsequent time you drag on another object it will default to the same object type as the previous one added.

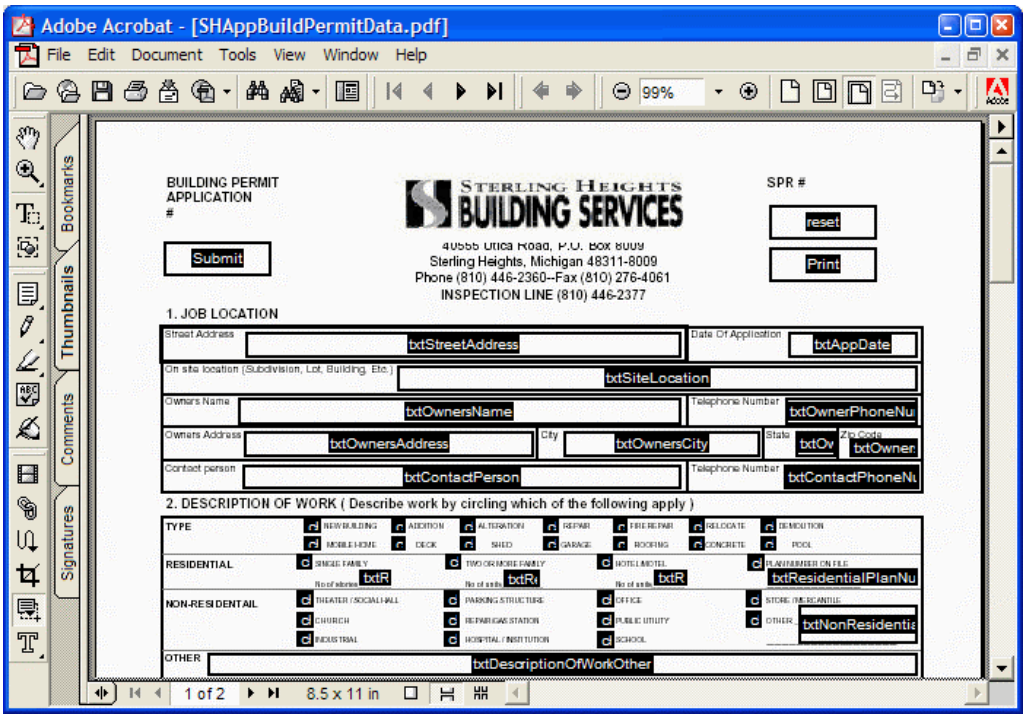


***Figure 12**. This is the same form, but now seen in "designer mode."*

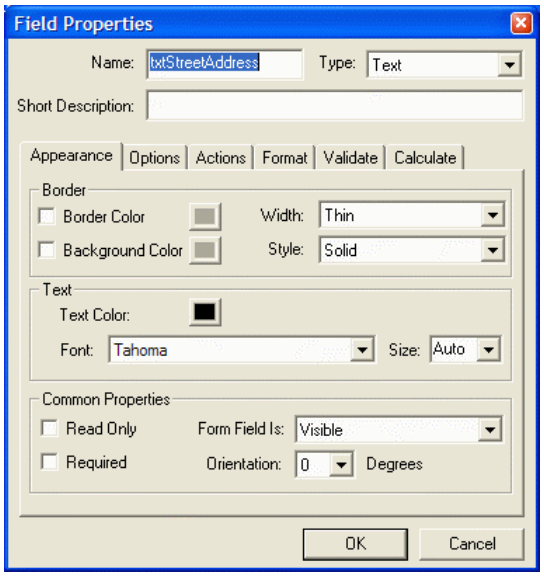


***Figure 13**. This is the Appearance page on the Acrobat Form Object Property Sheet for a Text object.*

Implementation of a PDF with Forms is identical to a regular PDF file. These files can be opened, data entered, and forms printed with the Reader version of Acrobat. The PDF can also be saved with the data included using the full version of Acrobat. The Amyuni product does not have any functionality concerning Acrobat Forms.

To this point we have not discussed the interaction with Visual FoxPro. The data captured in an Acrobat Form is exported via the File | Export | Form Data… menu option. This option is only available with the Business Tools or full Acrobat editions in version 4 of Acrobat. Version 5 requires the Approval or full Acrobat version. The export process creates a FDF file. This file is a flat text file that includes tags and data. Here is the information in the FDF file as it was exported from the SHBuildPermitData.pdf (included with the downloads):

```
%FDF-1.2
%âãÏÓ
1 0 obj
<<
/FDF << /Fields [ << /V /Off /T (chkNonResidentialTheater)>> << /V /Off /T
(chkResidentialChurch)>>
<< /V /Off /T (chkResidentialGasStation)>> << /V /Off /T
(chkResidentialHospital)>>
<< /V /Off /T (chkResidentialHotelMotel)>> << /V /Yes /T
(chkResidentialIndustrial)>>
<< /V /Off /T (chkResidentialOffice)>> << /V /Off /T (chkResidentialOther)>>
<< /V /Off /T (chkResidentialParkingStructure)>> << /V /Off /T
(chkResidentialPlanNumberOnFile)>>
<< /V /Off /T (chkResidentialPublicUtility)>> << /V /Off /T
(chkResidentialSchool)>>
<< /V /Off /T (chkResidentialSingle)>> << /V /Off /T (chkResidentialStore)>>
<< /V /Off /T (chkResidentialTwoOrMore)>> << /V /Off /T (chkTypeAddition)>>
<< /V /Off /T (chkTypeAlteration)>> << /V /Off /T (chkTypeConcrete)>>
<< /V /Off /T (chkTypeDeck)>> << /V /Off /T (chkTypeDemolition)>>
<< /V /Off /T (chkTypeFireRepair)>> << /V /Off /T (chkTypeGarage)>>
<< /V /Off /T (chktypeMobileHome)>> << /V /Yes /T (chkTypeNewBuilding)>>
<< /V /Off /T (chkTypePool)>> << /V /Off /T (chkTypeRelocate)>>
<< /V /Off /T (chkTypeRepair)>> << /V /Off /T (chkTypeRoofing)>>
<< /V /Off /T (chkTypeShed)>> << /V (11/15/2001)/T (txtAppDate)>>
<< /V (12/31/2099)/T (txtContactorHomeExpirationDate)>> << /V (38-9999999)/T
(txtContactorHomeFedId)>>
<< /V (987654321)/T (txtContactorHomeLicenseNumber)>> << /V (VFP Specialists)/T
(txtContactorHomeLicenseType)>>
<< /V (Sterling Heights)/T (txtContactorHomeOwnerCity)>> << /V (48313)/T
(txtContactorHomePostalCode)>>
<< /V (MI)/T (txtContactorHomeState)>> << /V (5865551234)/T
(txtContactorHomeTelephoneNumber)>>
<< /V (Weasel and Shifty Insurance Group, Inc.)/T
(txtContactorHomeWorkerCompIns)>>
<< /V (Steve Bodnar, Steve Sawyer, or Rick Schummer)/T (txtContactPerson)>>
<< /V (3134181290)/T (txtContactPhoneNumber)>> << /V (Acme Construction)/T
(txtContractorHomeOwner)>>
<< /V (9999 Elms Street)/T (txtContractorHomeOwnerAddress)>> << /V (New Geeks
and Gurus norther Detroit office)/T (txtDescriptionOfWorkOther)>>
<< /V (D37263128736218 78)/T (txtDriverLicense)>> << /V (999999999999)/T
(txtMESC)>>
<< /V (5869400081)/T (txtOwnerPhoneNumber)>> << /V (42424 Front Street)/T
(txtOwnersAddress)>>
```

```
<< /V (Sterling Heights)/T (txtOwnersCity)>> << /V (Geeks and Gurus, Inc.)/T
(txtOwnersName)>>
<< /V (48314)/T (txtOwnersPostalcode)>> << /V (MI)/T (txtOwnersState)>>
<< /V (Downtown Sterling Heights)/T (txtSiteLocation)>> << /V (9876 Main
Street)/T (txtStreetAddress)>>
]
/F (SHAppBuildPermitData.pdf)/ID [
<8c562dff8dbc2284ab14a9e4b572b02f><98995e30afea0090038a1c9c79587e1d>
] >>
>>
endobj
trailer
<<
/Root 1 0 R

>>
%%EOF
```

At this point we can see that the data entered can be output to a flat file. This file can be parsed using Visual FoxPro's Low-Level File Input and Output commands and added to tables, which are much easier for us to process. It would require that some fundamentally mundane code be written to separate the information from the tags and to get this information into a table. While most of us would not mind writing this code, wouldn't it be cool if there were a better mechanism to extract the data from the FDF format? There is, and it is called the FDF Toolkit, from Adobe.

How can I extract data out of a PDF form file? (*Example: FDFRead.prg*)

So now that we understand Acrobat PDF files can be built as a data entry mechanism and provide printing capability, the question begs, how do we extract this data from an Acrobat form and have it interact with our custom database applications? Adobe has provided a product called the FDF Toolkit on its Web site (http://partners.adobe.com/asn/developer/acrosdk/forms.html). This is a free product with a version for Acrobat 4 and 5 (our experience is that the version for 4 works with Acrobat 5, it just has fewer features). The download includes Application Programming Interfaces (API) for C/C++, Java, Perl, and ActiveX, and some extensive documentation on how it can be used with these tools. Visual FoxPro developers will find the Win32 ActiveX interface of the FDF Toolkit easy to use and very compatible (despite the lack of Visual FoxPro examples in the documentation). The ActiveX portion of the toolkit is made up of two files: FDFACX.DLL and FDFTK.DLL. The toolkit will install the toolkit files, but does not register the components.

The examples to read and write a FDF file will seem very familiar if you have worked with any Automation to Microsoft Word and the Visual FoxPro Low-Level File Input/Ouput commands (LLFIO). The example code can be found in the FDFREAD.PRG and FDFWRITE.PRG samples, which can be downloaded from Hentzenwerke.

Register the FDF Toolkit ActiveX control

The ActiveX control (FDFACX.DLL and corresponding FDFTK.DLL) should reside in the Windows/System32 directory or another directory that has "execute" permission. The process

to register the FDF Toolkit ActiveX control is as simple as the following command (add a path to the DLL if necessary):

```
RegSvr32 FdfAcX.dll
```

The control is self-registering. The Visual FoxPro 6 Setup Wizard and Visual FoxPro 7 InstallShield Express products will automatically register this control as part of the installation process so the deployment process is easy. Please note that there is no reason to register the FDFTK.DLL and that it will fail if you try to do so.

Instantiating the object to access the FDF File

The instantiation of the FDF ActiveX interface is accomplished via a standard process of using the Visual FoxPro **CREATEOBJECT()** function. Here is an example of the needed code:

```
loFDF = CREATEOBJECT("fdfApp.FdfApp")
```

This returns an object reference to the FDF control so that the methods can be run to read and write data from the FDF file. Now that we have the important object reference to the FDF control, we can start to manipulate the data inside of it via the interface methods that are exposed.

The first step in reading the information is to open the FDF file. This is accomplished by running the *FDFOpenFromFile()* method.

```
loFDFFile = loFDF.FDFOpenFromFile("SHAppBuildPermitData.fdf")
```

This method returns an object reference to the FDF file. If the file does not exist or could not be opened, an OLE Exception is thrown. You will need to handle this issue in your error-handling scheme. Once the object reference is gained you can go after specific fields in the FDF. To take this approach you need to provide the field name as a parameter to the *FDFGetValue()* method. One important item to note is the field names in the FDF, and access to these fields is case-sensitive. The passing of "txtstreetaddress" is not the same as "txtStreetAddress". So, to access a specific field you can use code like:

```
lcFDFField   = "txtStreetAddress"
luFieldValue = loFDFFile.FDFGetValue(lcFDFField)
```

You can also use the *FDFNextFieldName()* method to loop through the fields. To get the first field in the file you pass a null string (**SPACE(0)**) as the parameter to the *FDFNextFieldName()* method. To get the next field in the FDF file you pass the current field. Here is some code that loops through all the fields in the FDF file:

```
IF VARTYPE(loFDFFile) = "O"
  * Get the first field name in the FDF file
  lcFDFField     = loFDFFile.FDFNextFieldName("")
  lnFieldCounter = 1

  CLEAR
```

```
  * Loop through the FDF file to get the values
  DO WHILE NOT EMPTY(lcFDFField)
    luFieldValue   = loFDFFile.FDFGetValue(lcFDFField)

    ? str(lnFieldCounter, 6), lcFDFField, ;
      "(", vartype(luFieldValue), ") ==", luFieldValue

    lcFDFField = loFDFFile.FDFNextFieldName(lcFDFField)
    lnFieldCounter = lnFieldCounter + 1
  ENDDO
ENDIF

loFDFFile.FDFClose()
```

The data in the FDF file is strictly character-based. If you are moving this data into a table, you will likely need to transform the data into the proper data type for the field unless the record is all character fields.

There are hundreds of thousands of paper-based forms already pre-built, and a large percentage of these are already scanned and available on the Internet in PDF format. The examples used in this chapter were directly downloaded from the Sterling Heights city Web site. Many of the governmental and private business entities already have the forms set up in PDF format, and some are already set up with the form fields included. All the object fields were added in the example PDFs in less than 45 minutes. We did not add any JavaScript for serious validation or enforce any business rules in the examples, but it can be done with a little more effort. Leveraging existing PDF forms will save you time, your clients' money, and can make you look like the hero.

These forms can be used in a traditional LAN/workstation-based application as well as the client/server arena. The users open up Acrobat Reader and fill in the data in the form and use the menu to save the data to a predefined directory. Each user will need a full license to Acrobat (unless the new and less expensive Acrobat Approval meets your requirements). They will use the menu since the product does not support the JavaScript code necessary to export the data. In a Web site configuration the users open up the PDF in the browser and fill in the data. The Reader version (as well as the full version of Acrobat) can submit form data back to the Web server with JavaScript. We included a Submit button in the SHAPPBUILDPERMIT.PDF example to show the simple JavaScript code needed to submit the data back to the Web server. The data submitted from an Acrobat form is sent to the Web server in the same exact format as the data submitted from an HTML form. This information can be processed by a Common Gateway Interface (CGI) process. We have used WebConnect (from West Wind) to be the CGI process that accepts data from a PDF on the Web. The great thing about WebConnect in this situation is that it is extremely fast, and it allows Visual FoxPro developers to leverage their Visual FoxPro skills to provide a powerful solution.

How do I prefill the PDF Form with data? (*Example: FDFWrite.prg*)

Reading the file might be enough excitement for some of our clients, but what if they could also prefill a PDF Form with data from their Visual FoxPro application? The FDF Toolkit control also provides a plethora of methods to write out data into the FDF format. Once the object reference to the FDF ActiveX control is obtained, you execute the *FDFCreate()* method. This creates the FDF in memory and returns an object reference to this "file." After

the file is created, the field name tag (/F) and value tag (/V) are written for each of the fields you want written via the *FDFSetValue()* method. The following example writes out two fields:

```
loFDFFile      = loFDF.FDFCreate()

* Fill in two fields in the FDF
lcFDFField      = "txtStreetAddress"
lcFDFFieldValue = "1002 MegaFox Demo Street"
luFieldValue    = loFDFFile.FDFSetValue(lcFDFField, lcFDFFieldValue, .F.)

lcFDFField      = "txtOwnersName"
lcFDFFieldValue = "Enter your Name Here"
luFieldValue    = loFDFFile.FDFSetValue(lcFDFField, lcFDFFieldValue, .F.)
```

Naturally the code you will write will include more than a couple of fields. You also need to transform data from the native format to character before storing it in the FDF file. The final method called before closing the file is the *FDFSetFile()*. This writes out the /F tag, which associates the FDF file with the PDF file the data will be prefilled and display in. When the FDF file is opened it will preload the associated PDF file, and then fill in the fields loaded in the FDF.

```
* Set the name of the PDF associated with the FDF
loFDFFile.FDFSetFile("SHAppBuildPermitForm.pdf")
```

The *FDFSaveToFile()* physically writes out the FDF data to a file. The file is closed via the *FDFClose()* method and the object reference should be released.

```
* Write out the file
loFDFFile.FDFSaveToFile("Chapter08Sample.fdf")
loFDFFile.FDFClose()
```

There are a number of other methods in the FDF ActiveX that provide behaviors you may find useful. There are capabilities to write FDF files to a string, additional tags can be inserted into the file, and you can add custom JavaScript, among other things.

There are two real-life examples using the FDF Toolkit to prefill data in a PDF form that we would like to discuss. The first is to use it as a substitution of the Visual FoxPro Report Designer. Customers are always demanding reports that replicate the paper forms. Some of these reports can be quite challenging using the Visual FoxPro Report Designer or any third-party reporting tool. Since we can see that plugging data into a PDF can be straightforward, why not take advantage of this technique? Generate the FDF reference, plug in the data, and save it to a temporary file. Using the techniques discussed in the section "How can I replace the Visual FoxPro Report print preview?" you can shell Acrobat Reader for the users to preview the report, and they can print it using the Reader interface, or via a button like we included in the SHAPPBUILDPERMIT.PDF example. You can also display the PDF file in a Visual FoxPro form and manipulate it via the ActiveX interface.

The second example is to place the PDF on a Web site or in a custom application for data entry. If there is default data that can be plugged into the PDF form from the application's database, use the FDF Toolkit to plug in the data before the user sees the PDF in the reader. We do this with our Visual FoxPro forms all the time; why should using this interface be

different? On the Internet you will return the FDF file to the browser, which will instance the Acrobat ActiveX control based on the file association of the FDF. The Acrobat control will request the PDF file from the Web server and the PDF will be displayed with data prefilled in the browser.

How can I merge PDF files together? (*Example: PDFMerger.prg/PDFDirectoryMerger.prg*)

This chapter has demonstrated a number of ways to generate PDF files from Visual FoxPro reports. There are times when merging different reports together into one PDF file is a requirement of the customer. This section will discuss one way to accomplish merging two PDFs together using ActiveX components provided with the full version of Adobe Acrobat, and then demonstrate how a complete directory of PDF files can be merged into one (see **Listing 2**).

Acrobat has an ActiveX interface. First you instantiate a reference to the AcroExch.App object and an object reference to AcroExch.PDDoc for each of the PDF files that you want merged together. The *Open()* method of the AcroExch.PDDoc opens the PDF file and establishes an object reference to the PDF. The *GetNumPages()* method returns the number of pages in the PDF. It should be noted that the number of pages in the PDF file returned from the *GetNumPages()* is zero-based (starts at zero).

The actual merging of the files happens with the *InsertPages()* method. The first parameter is the page number that you want the merge to start after. Typically you will merge after the last page, but you can insert a PDF anywhere in another PDF. The second parameter is an object reference to the second PDF file via the AcroExch.PDDoc object. The third parameter is the start page. Again, the internal page numbers in a PDF file start with zero, so if you want to get the first page you would pass a zero. The fourth parameter is the number of pages to insert. The last parameter indicates whether you also want the bookmarks inserted as well.

***Listing 2**. Partial code listing of PDFMerger.prg, which demonstrates how to merge two PDF files together.*

```
LPARAMETERS tcPDFOne, tcPDFTwo, tcPDFCombined, tlShowAcrobat

#DEFINE  ccSAVEFULL 0x0001

LOCAL loAcrobatExchApp, ;
      loAcrobatExchPDFOne, ;
      loAcrobatExchPDFTwo, ;
      lnLastPage, ;
      lnNumberOfPagesToInsert, ;
      lcOldSafety

lcOldSafety = SET("Safety")
SET SAFETY OFF
ERASE tcPDFCombined
SET SAFETY &lcOldSafety

* Get appropriate references to Acrobat objects
loAcrobatExchApp    = CREATEOBJECT("AcroExch.App")
loAcrobatExchPDFOne = CREATEOBJECT("AcroExch.PDDoc")
```

```
loAcrobatExchPDFTwo = CREATEOBJECT("AcroExch.PDDoc")

* Show the Acrobat Exchange window
IF tlShowAcrobat
   loAcrobatExchApp.Show()
ENDIF

* Open the first file in the directory
loAcrobatExchPDFOne.Open(tcPDFOne)

* Get the total pages less one for the last page num [zero based]
lnLastPage = loAcrobatExchPDFOne.GetNumPages() - 1

* Open the file to insert
loAcrobatExchPDFTwo.Open(tcPDFTwo)

* Get the number of pages to insert
lnNumberOfPagesToInsert = loAcrobatExchPDFTwo.GetNumPages()

* Insert the pages
loAcrobatExchPDFOne.InsertPages(lnLastPage, loAcrobatExchPDFTwo, 0, ;
                                lnNumberOfPagesToInsert, .T.)

* Close the document
loAcrobatExchPDFTwo.Close()

* Save the entire document, saved as file passed as third
* parameter to program using SaveFull [0x0001].
loAcrobatExchPDFOne.Save(ccSAVEFULL, tcPDFCombined)

* Close the PDDoc
loAcrobatExchPDFOne.Close()

* Close Acrobat Exchange
loAcrobatExchApp.Exit()

* Need to release the objects
RELEASE loAcrobatExchPDFTwo
RELEASE loAcrobatExchPDFOne
RELEASE loAcrobatExchApp

WAIT CLEAR

RETURN SPACE(0)
```

Acrobat will not merge secured PDF documents. The result of a merge between one secure PDF document and a non-secure PDF document will be the contents of the non-secure PDF document. ***Figure 14*** *shows the Document Security screen, which details the security settings for a PDF file.*

***Figure 14**. The Acrobat Document Security screen (File | Document Security… menu) will inform you of the security settings for the PDF file.*

We have found the performance of the merge functionality to be very snappy. We have merged small PDFs (10KB) with large PDFs (over 1MB), and large PDFs with other large PDFs in a couple of seconds or less. The merge process will also merge the bookmarks in one or both documents.

The merge process is useful when merging in a number of different Visual FoxPro reports to build an executive package. You can also merge in PDFs generated from other applications like Word, Excel, or other custom Visual FoxPro applications. The source of the PDF files or the method used to create the PDF does not matter. One example of this could be a header page template generated from Word with some nice graphics and some fancy fonts. Merge in an introductory letter created in Word and saved to a PDF. The next few pages could be a Visual FoxPro report that outlines sales figures for the region. Merge in some nice graphs that were generated via Automation from the Visual FoxPro custom application to Excel and printed to a PDF. The last merge could be another summary from the Sales Manager created in Word and saved to a PDF file. There is no limitation to the merging other than file size and the amount of disk space.

If you want to merge in a number of PDF files in a directory, you can use code that calls the PDFMerger program. Here is a partial listing of PDFDIRECTORYMERGER.PRG:

```
DIMENSION laPDFFiles[1]

lcFileSkeleton = ADDBS(ALLTRIM(tcDirectory)) + "*.pdf"
lnPDFCount     = ADIR(laPDFFiles, lcFileSkeleton)

DO CASE
  CASE lnPDFCount > 1
    lcLastFile = tcDirectory + laPDFFiles[1, 1]

    FOR lnCount = 2 TO lnPDFCount
      IF lnCount = lnPDFCount
        * Last one, used the specified combine file
        lcCombinedFile = tcPDFCombinedFile
```

```
      ELSE
        * Build a temporary
        lcCombinedFile = FORCEEXT(ADDBS(SYS(2023)) + "Temp" + ;
                                  ALLTRIM(STR(lnCount)), "PDF")
      ENDIF

      lcResult   = PdfMerger(lcLastFile, ;
                             tcDirectory + laPDFFiles[lnCount, 1], ;
                             lcCombinedFile)
      lcLastFile = lcCombinedFile
    ENDFOR

  CASE lnPDFCount = 1
    COPY FILE laPDFFiles[1, 1] TO tcPDFCombinedFile

  OTHERWISE
    * Nothing to do with no files in directory
ENDCASE
```

The program loops through all the PDF files in the specified directory and merges them into one file (based on a parameter passed to the program).

Conclusion

This chapter demonstrates a number of ways to integrate Adobe Acrobat technology with custom Visual FoxPro applications. The ideas presented show alternative methods of generating reports, e-mailing report output, displaying reports in preview mode without the Visual FoxPro report preview limitations, and capturing information from the users and presenting the same information using Acrobat Forms. We hope you enjoyed reading it and that you have some idea how to integrate the power of Acrobat PDF technology with your custom Visual FoxPro applications.

Chapter 9
Using ActiveX Controls

ActiveX controls have been around for quite a while now, and are quite widely used by developers working in other languages. However, they have never been really popular among FoxPro developers. This has always struck us as a shame because there are some very good ActiveX controls available, completely free, which provide useful functionality with little or no effort. In this chapter we will show you how you can leverage some of these standard controls to extend your Visual FoxPro applications.

How do I include ActiveX controls in a VFP Application?

ActiveX controls are distributed as files with an .OCX extension, and quite a number of them are used by Windows and are, therefore, already installed on your system. More ship with Visual FoxPro (see **Table 1**), and yet more are available from third-party suppliers and vendors. One common problem with third-party ActiveX controls is that Visual FoxPro implements the ActiveX interface guidelines rigorously and, apparently, much more rigorously than some other tools. The result is that controls that have been written for and tested in, say, Visual Basic, simply don't work in Visual FoxPro at all.

***Table 1**. ActiveX controls, and their OCX files, that ship with VFP 7.0.*

Control	File	Help file
Animation control	MSCOMCT2.OCX	CMCTL298.CHM
Datetimepicker control	MSCOMCT2.OCX	CMCTL298.CHM
ImageCombo control	MSCOMCTL.OCX	CMCTL198.CHM
ImageList control	MSCOMCTL.OCX	CMCTL198.CHM
ListView control	MSCOMCTL.OCX	CMCTL198.CHM
MAPI Message control	MSMAPI32.OCX	MAPI98.CHM
MAPI Session control	MSMAPI32.OCX	MAPI98.CHM
Masked Edit control	MSMASK32.OCX	MASKED98.CHM
Microsoft Internet Transfer control	MSINET.OCX	INET98.CHM
Monthview control	MSCOMCT2.OCX	CMCTL298.CHM
MsChart control	MSCHRT20.OCX	MSCHRT98.CHM
MsComm control	MSCOMM32.OCX	COMM98.CHM
Multimedia MCI control	MCI32.OCX	MMEDIA.CHM
PicClip control	PICCLP32.OCX	PICCLP98.CHM
ProgressBar control	MSCOMCTL.OCX	CMCTL198.CHM
Rich Textbox control	RICHTX32.OCX	RTFBOX98.CHM
Slider control	MSCOMCTL.OCX	CMCTL198.CHM
StatusBar control	MSCOMCTL.OCX	CMCTL198.CHM
SysInfo control	SYSINFO.OCX	SYSINF98.CHM
TabStrip control	MSCOMCTL.OCX	CMCTL198.CHM
Toolbar control	MSCOMCTL.OCX	CMCTL198.CHM
TreeView control	MSCOMCTL.OCX	CMCTL198.CHM
Updown control	MSCOMCT2.OCX	CMCTL298.CHM
Visual FoxPro Foxtlib control	FOXTLIB.OCX	FOXHELP.CHM
Winsock control	MSWINSCK.OCX	MSWNSK98.CHM

You can be reasonably confident that the controls listed in Table 1 have been tested with Visual FoxPro and will, at least under ideal conditions, work as advertised. However, the only way you can be certain about the compatibility of any other control is to try it. Having said that, there are plenty of good ActiveX controls available that can greatly enhance the appearance and functionality of your applications.

One other point that you need to be aware of is that care is needed when distributing ActiveX controls. This is because, unfortunately, ActiveX controls suffer from the same lack of cross-version compatibility and control issues as any other DLL. The basic problem is illustrated by **Table 2**, which shows how the ActiveX control named "CoolTool"—which, was originally shipped as a file named ACX01.OCX—has been amended over time. The first new version of the control extends its interface, but the file name remains the same. This gives us a problem when an application that specifically installs Version 1.0 is installed (or re-installed!) on a system after that system has already been upgraded to Version 2.0. The newer file is simply overwritten and the extended interface is lost with potentially disastrous consequences.

***Table 2**. The ActiveX version control problem.*

Version	File name	Class name	Methods	ProgID
1.0	ACX01.OCX	CoolTool	Start, Stop	ACX.CoolControl.1
2.0	ACX01.OCX	CoolTool	Start, Stop, Suspend	ACX.CoolControl.2
3.0	ACX02.OCX	CoolTool	Strat, Stop, Suspend, Resume	ACX.CoolControl.3

In an attempt to avoid this problem, the name of the file that was distributed was often changed for new versions although the class name must still remain the same irrespective of the file name. This is illustrated by the Version 3.0 release of CoolTool, which is shown as having been shipped as ACX02.OCX. It is then possible to have both Version 2 and Version 3 files installed on the same machine.

However, there is still only one class name, and the problem now is that whichever file was registered last is the one with which the Registry will associate the class name. So even though the correct file may exist, there is no guarantee that it will always be used to instantiate the most recent version of the control. The reason for this is the way that the Registry stores information about ActiveX controls.

Each control has a unique identifier that is used as the Class ID key for that control. Associated with the Class ID is the actual name of the control and several other vital pieces of information.

Figure 1 shows the Class ID entry for the "Microsoft Date and Time Picker Control 6.0 (SP4)" ActiveX control. Note that there are both a *ProgID* (whose value is "MSComctl2.DTPicker.2") and a *VersionIndependentProgID* (whose value is "MSComctl2.DTPicker"). Unfortunately, when you create a subclass of an ActiveX control, Visual FoxPro always inserts the ProgID into the *OleClass* property, which means that any control created in this way by Visual FoxPro is version-specific. In fact, the only way you can avoid using the ProgID is to add the ActiveX control programmatically at run time (see "How do I add an ActiveX control to a form or class?").

By default, OCX files are installed in *System32* under the Windows home directory, but they can also be installed in your application's home directory along with the EXE file itself, or even better, to an "*application common*" directory (for more specific information on

deploying applications that include ActiveX controls, see Chapter 11). To include an ActiveX control in your application, simply include the parent OCX file in your project.

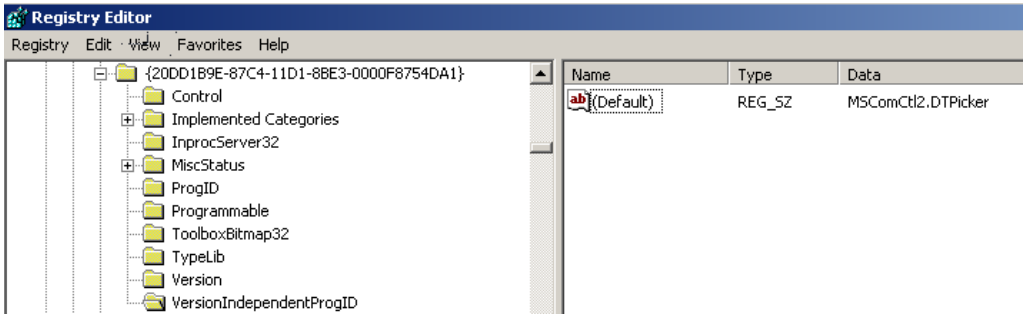


***Figure 1**. Registry entry for ActiveX Date and Time picker.*

How do I find out what controls are in an OCX?

As Table 1 shows, each of the OCX files that ships with Visual FoxPro has an associated Help file that usually gives a lot of useful information. However, do remember that the Visual FoxPro Object Browser will open the type library associated with an OCX just as easily as one for a DLL and it provides a quick and easy way to research the contents of an OCX. More importantly, you can also get the actual values for the constants from the Object Browser. **Figure 2** shows the Object Browser after loading in the "Microsoft Progress Bar Control (SP4)," which is one of several controls contained in MSCOMCTL.OCX.

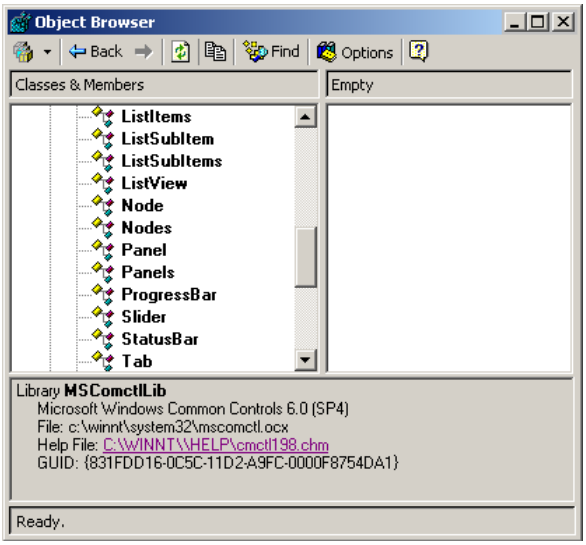


***Figure 2**. MSComCtl.ocx in the Object Browser.*

Okay, but how do I get the class name of an ActiveX control?

That is a very good question! This one caused us considerable grief, as we couldn't actually find the answer anywhere in the Visual FoxPro documentation. One certain way is to create an instance of the desired control visually in a form and inspect its *OLEClass* property. Finally, after much digging, we also found an article in the MSDN Knowledge Base (*Q191222*) that lists, for Visual FoxPro Version 6.0, the class names for the ActiveX controls that ship with the product. **Table 3** lists the class names and version numbers (that is, the ProgID) as defined in that article.

***Table 3**. Class names for the ActiveX controls.*

File	Controls	OleClass
MSCOMCT2.OCX	Animation control	MSComCtl2.Animation.2
	DateTimePicker control	MSComCtl2.DTPicker.2
	MonthView control	MSComCtl2.MonthView.2
	UpDown control	MSComCtl2.UpDown.2
SYSINFO.OCX	SysInfo control	SysInfo.SysInfo.1
RICHTX32.OCX	Rich Textbox control	RichText.RichTextCtrl.1
PICCLP32.OCX	PicClip control	PicClip.PictureClip.1
MSWINSCK.OCX	WinSock control	MSWinsock.Winsock.1
MSMASK32.OCX	Masked Edit control	MsMask.MaskEdBox.1
MSMAPI32.OCX	MAPI Message control	MSMAPI.MapiMessage.1
	MAPI Session control	MSMAPI.MapiSession.1
MSINET.OCX	Microsoft Internet Transfer control	InetCtls.Inet.1
MSCOMM32.OCX	MSComm control	MSCommLib.MSComm.1
MSCOMCTL.OCX	ImageCombo control	MSComCtlLib.ImageComboCtl.2
	ImageList control	MSComCtlLib.ImageListCtrl.2
	ListView control	MSComCtlLib.ListViewCtrl.2
	ProgressBar control	MSComCtlLib.ProgCtrl.2
	Slider control	MSComCtlLib.Slider.2
	StatusBar control	MSComCtlLib.SBarCtrl.2
	Toolbar control	MSComCtlLib.Toolbar.2
	TreeView control	MSComCtlLib.TreeCtrl.2
MSCHRT20.OCX	MsChart control	MSChartLib.MSChart.2
MCI32.OCX	Multimedia MCI control	MCI.MMControl.1

Note that although the version number is shown as part of the class name, if it is not specified Windows will use the first version of the control that it can find. So, unless you use features of a control that are version-specific, there is no need to include the version number when instantiating an ActiveX control. However, if you *do* make use of version-specific features, you must ensure that the correct version of the OCX file is shipped with, and installed by, your application (as noted earlier, the best solution is to install such OCX files in a directory that is explicitly referenced in your application's search path).

How do I add an ActiveX control to a form or class?

There are several ways of adding an ActiveX control in either of the visual designers. First, you can drop an instance of the *OLEContainer* control base class on to the design surface. This will pop up a dialog that allows you to specify the control you wish to use. Simply ensure that the Insert Control option is selected and click the OK button to create the control.

Alternatively, you can use the Controls tab of Visual FoxPro's Options dialog to view, and select from, a list of all registered controls. (This subset can be made persistent by clicking the Set As Default button.) To access these controls, select the ActiveX Controls option from the Controls toolbar in the appropriate designer. You can now select any of the controls you have previously specified and drop it directly onto your design surface.

Finally, you can create an instance of an ActiveX control programmatically. All that is needed is to add an instance of the *OLEControl* base class and set its *OleClass* property to point to the required control or, if you are using the *AddObject()* method, you can pass the ActiveX control name as a parameter, like this:

```
*** To add the Status Bar ActiveX Control to a form
WITH ThisForm
  .AddObject( 'oAX', 'olecontrol', 'MSComctlLib.sBarCtrl' )
  WITH .oAX
    WITH .Panels(1)
      .TEXT = "Sample Text"
      .TOOLTIPTEXT = "Panel 1"
      .STYLE = 0
    ENDWITH
    .Height = 25
    .Visible = .T.
  ENDWITH
ENDWITH
```

One benefit of doing this programmatically at run time is that you can utilize the VersionIndependentProgID and so avoid some of the versioning issues associated with using the visual designers noted earlier in this chapter.

We have always strongly advocated the creation of "buffer" subclasses between the native VFP base classes and your own classes to minimize the impact of changes in class definitions at the product level. Given the known issues with versioning of ActiveX controls, these subclasses are even more important than usual and, even for third-party controls, the first thing you should do is to create your own subclasses so that you are always working with a known version. Of course, there are other benefits to doing this—you can also set your own preferences for defaults and add custom properties and/or methods.

Putting ActiveX controls to use

The remainder of this chapter is devoted to a review of the main ActiveX controls that ship with Visual FoxPro Version 7.0. The objective here, as always, is to provide a working example for each control without simply duplicating the information in the Help file. However, it is impossible to cover all the options and permutations for these controls, so the task of exploring, in detail, any control that is of special interest remains your own. We'll start with a simple example and take things from there.

How do I subclass an ActiveX control? (*Example: CH09.vcx::xprogbar*)

The easiest way to illustrate this is to create a visual class (although you can, of course, do the same thing in code if you wish). Start by defining a new class based on an OLE Container control. (This, by the way, is one of the rare exceptions where we do not bother with a custom "root class" but simply use the Visual FoxPro base class for an *OleContainer* control directly.)

When you do this in the visual class designer you will (after a moment) see a dialog that lists all of the available ActiveX controls. Select the "Microsoft Progress Bar Control 6.0" and click the OK button to add the control. That is all that there is to it. All that is left to do is to set any class-level properties that you wish, and add any custom properties/methods.

Note that this control, like most (but not all!) ActiveX controls, has *two* ways of accessing its properties. First, you can use the normal Visual FoxPro properties sheet, which shows, on the individual Data, Methods, Layout, and Other tabs, only the PEMs for the *OleContainer* control. The All tab, however, also shows any PEMS defined by the contained ActiveX control. (You can see this by checking for properties named Max, Min, and Value.)

This is not a particularly good way of setting the control's properties since, unless you know in advance what PEMs the control is exposing, there is no easy way to find them. Fortunately, there is a simpler way. If you right-click on the control in the design surface you will notice that a new option has appeared in the pop-up menu entitled "ProgCtrl Properties". This brings up a property sheet (see **Figure 3**) that contains only those PEMs that are exposed by the ActiveX control itself. This is generally the better way to set the properties for ActiveX controls, although either way will work.

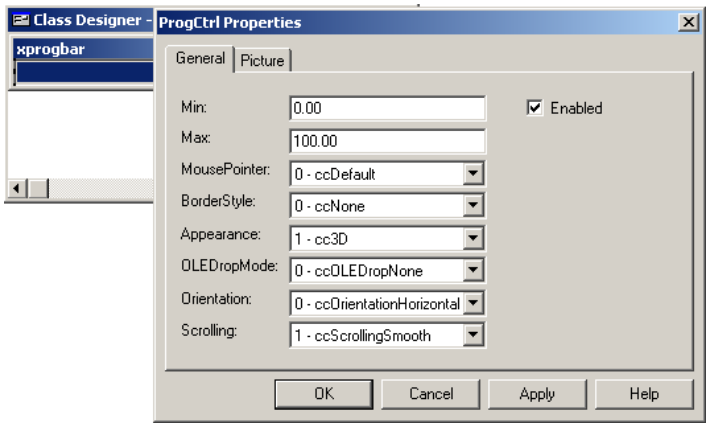


***Figure 3**. Property sheet for the progress bar ActiveX control.*

How do I use the Windows progress bar? (*Example: CH09.vcx::xTherm; frmprogbar.scx*)

You can see from Table 3 that the progress bar control is actually a class named "MSComCtlLib.ProgCtrl" that is contained in MSCOMCTL.OCX. We created, exactly as described earlier, a subclass of this control in our CH09.VCX visual class library named "xTherm".

If you examine this class, you will notice that the *OleClass* property (filled in when we selected the control from the dialog) includes the version number. Now, we did say that you should, where possible, avoid specifying the version number. However, even if you edit the class to remove the version number (you need to open the class library as a table and edit the *Properties* memo field directly to do this) it still appears in the property sheet. This is because the properties sheet shows the full name of class that was actually used to create the control, not merely the name that was defined in your class.

Setting up the progress bar class

The Progress Bar control, as illustrated in Figure 3, exposes several properties. However, for the moment we are only interested in three of them:

- *Min* — Defines the lower limit (0%) of the range represented by the progress bar.
- *Max* — Defines the upper limit (100%) of the range represented by the progress bar.
- *Scrolling* — Defines the appearance of the progress bar; possible values are either "0 ccScrollingStandard" (bar is a series of discrete blocks) or "1 ccScrollingSmooth" (bar is continuous).

In addition, the control has a *Value* property whose content actually determines how much of the progress bar is displayed. We intend to show this class working on a "percentage complete" basis so, for now, we can leave the *Min* and *Max* properties at their default values. However, to avoid errors, we have added a custom assign method to the *Value* property to ensure that any specified value falls into the defined range, as illustrated here:

```
LPARAMETERS tnNewVal
IF VARTYPE( tnNewVal ) # "N" OR EMPTY( tnNewVal ) OR tnNewVal < This.Min
  *** Set to Min Value if invalid, nothing or less than Min
  lnPCDone = This.Min
ELSE
  *** Force to Max Value if greater than Max
  lnPCDone = IIF( BETWEEN( tnNewVal, This.Min, This.Max ), tnNewVal, This.Max )
ENDIF
*** Update the display
This.Value = lnPCDone
```

The only other property we *need* to set is the *Scrolling* property. By default the progress bar shows a series of discrete blocks as the value is incremented toward the maximum (see **Figure 4**).

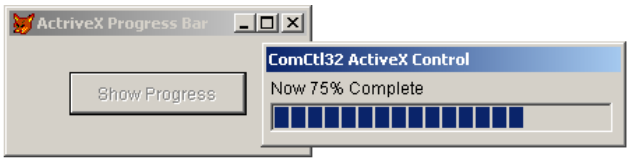


***Figure 4**. Standard progress bar display.*

However, by setting Scrolling = 1, we can display a continuous progress bar (see **Figure 5**). Our personal preference is for the smooth bar, so we have made this the default in our class. All other properties can safely be left at their default settings, although it is worth mentioning that the *Orientation* property can be set to display a vertical progress bar instead of the usual horizontal.

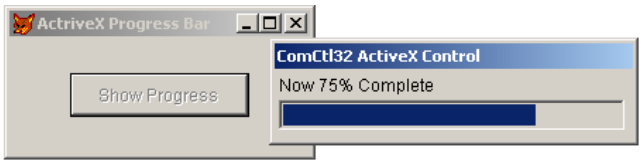


***Figure 5**. Smooth scrolling progress bar display.*

Displaying the progress bar

In order to actually display the progress bar, it must be added to an object that is capable of being made visible (that is, a form or a toolbar). To illustrate how it might be used, we have created a form class (*xTherm*) that can accept, in its *Init()* method, parameters that are used by the custom *SetLabels()* method to set the form's *Caption* and "comment" label:

```
LPARAMETERS tcLbl01, tcLbl02
*** Only change values if something is specified
WITH This
  IF PCOUNT() = 1
    *** Assume we just want to change the comment
    .lblPrompt.Caption = IIF( EMPTY( tcLbl01 ), "", ;
                              ALLTRIM( TRANSFORM( tcLbl01 )))
  ELSE
    IF PCOUNT() = 2
      .lblPrompt.Caption = IIF( EMPTY( tcLbl01 ), "", ;
                                ALLTRIM( TRANSFORM( tcLbl01 )))
      .Caption = IIF( EMPTY( tcLbl02 ), "", ;
                                ALLTRIM( TRANSFORM( tcLbl02 )))
    ENDIF
  ENDIF
ENDWITH
```

The form also has an exposed custom method named *UpdateTherm()* that accepts a numeric value. This value is used to change the amount of progress displayed. You can also pass this method a character string, as the second parameter, which is passed on to the *SetLabels()* method and so updates the "comment".

```
LPARAMETERS tnNewVal, tcNewPrompt
*** The value has an assign method, so just pass the value through 'as-is'
ThisForm.oBar.Value = tnNewVal
IF PCOUNT() = 2
  *** We got a caption too
  This.SetLabels( tcNewPrompt )
ENDIF
```

That is all the code that there is in the form class; the Show Progress button in the example form simply instantiates this class and then calls its *UpdateTherm()* method inside a loop as follows:

```
LOCAL loTherm, lnCnt
*** Create the progress form, and set its caption
loTherm = NEWOBJECT( 'xTherm','ch09.vcx','','','ComCtl32 ActiveX Control' )
loTherm.Visible = .T.
*** Update progress bar display and Comment
FOR lnCnt = 1 TO 100
  loTherm.UpdateTherm( lnCnt, "Now " + TRANSFORM( lnCnt ) + "% Complete" )
  INKEY(0.01,'h')
NEXT
RELEASE loTherm
```

This is, perhaps, the simplest of all the ActiveX controls, but it does illustrate the principles of using one, and shows how little code is actually required in order to make it perform. In fact, all we really need to do is to set its *Value* property; everything else is merely window-dressing.

How do I use the Date and Time Picker? (*Example: CH09.vcx::acxDTPicker and DateTimePicker.scx*)

The Date and Time Picker is similar to the calendar control we discussed in Chapter 1. However, it has one major limitation that the calendar control does not: It cannot be bound to empty or null dates. Having said that, it provides a much richer visual interface and is a lot more modern looking than the control contained in MSCAL.OCX (see **Figure 6**). As its name implies, The Date and Time Picker can be used to handle DateTime values as well as simple dates. By creating an intelligent subclass of the control, we can even get around the limitation of not being able to bind it to empty values.

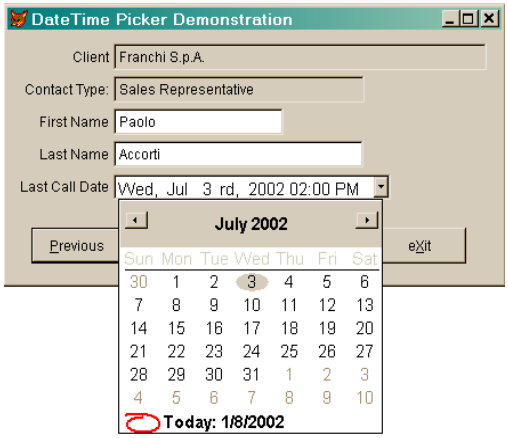


***Figure 6**. The Date and Time Picker in action.*

The Date and Time Picker exposes numerous properties that allow you to exert fine control over its appearance. They are reasonably well documented in CMCTL298.CHM, the Help

file for MSCOMMCT2.OCX. Properties like *CalendarForeColor*, *CalendarBackColor*, *CalendarTitleForeColor*, and *CalendarTitleBackColor* are self-explanatory and you should refer to the Help file for allowable settings.

Perhaps the most useful property is *Format*; it specifies how the data is displayed in the textbox portion of the control. There are three pre-defined formats that you can use or, by setting the *Format* to 3, you can specify your own format using the *CustomFormat* property (see **Table 4**).

***Table 4**. Possible formats for the Date and Time Picker.*

Value	Explanation
0	Long date as specified in the Windows control panel. For example: Sunday, July 25, 2002.
1	Short date as specified in the Windows control panel. For example: 7/25/02.
2	Time format. For example: 4:20 PM. Note that even though only the time is displayed, the control's value still contains the date portion of the value.
3	Custom. This setting allows you to specify your own custom format in the control's *CustomFormat* property.

When the Date and Time Picker's *Format* property is set to 3-Custom, you must specify the *CustomFormat*. This property defines the format expression that will be used to display the date in the textbox portion of the control. The format strings shown in **Table 5** are supported by the control.

***Table 5**. Possible custom formats for the Date and Time Picker.*

String	Description
d	The one- or two-digit day.
dd	The two-digit day. Single-digit day values are preceded by a zero.
ddd	The three-character day-of-week abbreviation.
dddd	The full day-of-week name.
h	The one- or two-digit hour in 12-hour format.
hh	The two-digit hour in 12-hour format. Single-digit values are preceded by a zero.
H	The one- or two-digit hour in 24-hour format.
HH	The two-digit hour in 24-hour format. Single-digit values are preceded by a zero.
m	The one- or two-digit minute.
mm	The two-digit minute. Single-digit values are preceded by a zero.
M	The one- or two-digit month number.
MM	The two-digit month number. Single-digit values are preceded by a zero.
MMM	The three-character month abbreviation.
MMMM	The full month name.
s	The one- or two- digit seconds.
ss	The two-digit seconds. Single-digit values are proceeded by a zero.
t	The one-letter AM/PM abbreviation (that is, "AM" is displayed as "A").
tt	The two-letter AM/PM abbreviation (that is, "AM" is displayed as "AM").
X	A callback field that gives programmer control over the displayed field. Multiple "X" characters can be used in series to signify unique callback fields.
y	The one-digit year. For example, 2002 would be displayed as 2.
yy	The last two digits of the year. For example, 2002 would be displayed as 02.
yyy	The full year. For example, 2002 would be displayed as 2002.

Notice the callback field, specified by the format string "X", in Table 5. This is what enabled us to display the suffix "rd" in sample form pictured in Figure 6. The use of callback fields enables us to customize the display in the textbox portion of the control to our heart's content. All we need to do is specify a string of X's in the control's *CustomFormat* for each callback field that we want to display. So, in order to display the correct suffix for the day number in our sample form, we used a *CustomFormat* of **`ddd, MMM dX yyy hh:mm tt`**.

When a *CustomFormat* that includes *CallBack* field is specified, the *Format* and *FormatSize* events are raised for each callback field whenever the control is refreshed. We can write code in the *Format* event to specify a custom response string. If this custom response string is of variable length, the *FormatSize* event is used to determine the space required to display the string. So, in order to display the correct suffix, we used this code in the *Format* event of our custom *acxDTPicker* class:

```
LPARAMETERS callbackfield, formattedstring
LOCAL lnNdx
*** Add the appropriate suffix to the date
IF CallBackField = 'X'
  lnNdx = This.Day % 10
  IF ( NOT BETWEEN( lnNdx, 1, 3 ) ) OR ( BETWEEN( This.Day, 11, 13 ) )
    FormattedString = 'th,'
  ELSE
    FormattedString = SUBSTR( 'stndrd', ( 2 * lnNdx ) - 1, 2 ) + ','
  ENDIF
ENDIF
```

We can even control how the Date and Time Picker responds to keyboard events when a *CallBack* field is selected. This is accomplished by using the control's *CallBackKeyDown()* method. As a matter of fact, if we want to increment a value when the **`UP ARROW`** key is pressed while in a *CallBack* field, the only way to do this is by intercepting it the *CallBackKeyDown()* method and taking appropriate action. This can be a real pain if we want our *CallBack* fields to behave like the rest of the fields in the control. Pressing the **`UP ARROW`** key in the day, month, or year fields of the textbox increments them, and we get this functionality for free.

We can also include strings in our custom format to display things like "Your appointment is on Wed, Jul 3rd, 2002 at 2:00 PM". All we have to do is specify the string literals, wrapped in quotes, right in the *CustomFormat* property; it's that easy.

So what is the CheckBox property for?

We stated earlier that it is not possible to bind the Date and Time Picker to an empty or null date. While this is true, the control's *CheckBox* property allows you to specify whether or not the control's *Value* is actually set to the displayed date.

When the *CheckBox* property of the Date and Time Picker is set to True, a small check box appears to the left of the display in the text box portion of the control. If the box is left unchecked, the control's *Value* property is null. When it is checked, the *Value* contains the displayed date. We think this is both confusing and user-surly. That is why we created our own custom subclass of the Date and Time Picker to get around the issue of null and empty dates.

How does the custom acxDTPicker class work?

We began the creation of our custom acxDTPicker class in the visual class designer in the usual way. We set up default values for the *Format* and *CustomFormat* properties using the ActiveX control's property sheet (see **Figure 7**), which can be displayed by selecting "DTPicker Properties" from the right-click shortcut menu in the design surface.

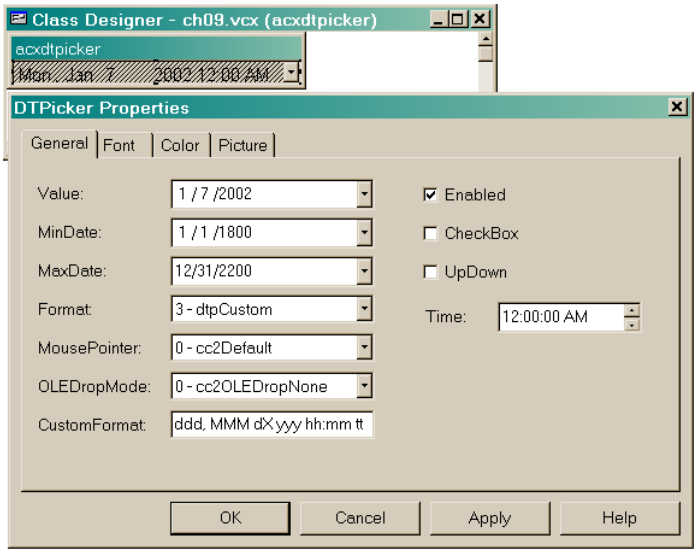


***Figure 7**. Date and Time Picker properties.*

The problem we had to solve was that we wanted to be able to bind the control to data in our forms but did not want it to throw an OLE error if the data happened to be either empty or null. One custom property and two custom methods are needed so that we can unbind the control but, at the same time, have it behave like a bound control.

The custom *cControlSource* property is used to store the control's original *ControlSource* before it is unbound in the *Init()* like this:

```
WITH This
  IF NOT EMPTY( .ControlSource )
    .cControlSource = .ControlSource

    *** If the first date is empty
    *** we must set it to something before unbinding the control
    *** otherwise, we STILL get the OLE error
    ldValue = EVALUATE( .ControlSource )
    lcField = JUSTEXT( .ControlSource )
    lcAlias = JUSTSTEM ( .ControlSource )
    IF EMPTY( ldValue )
      REPLACE ( lcField ) WITH DATETIME() IN ( lcAlias )
    ENDIF
    .ControlSource = ''
    REPLACE ( lcField ) WITH ldValue IN ( lcAlias )
  ENDIF
ENDWITH
```

One consequence of the approach discussed previously is that it dirties the buffers. This can easily be remedied by using **SETFLDSTATE()** like this:

```
lcFldState = GETFLDSTATE( -1, lcAlias )
IF '1' $ lcFldState
  SETFLDSTATE( lcField, 1, lcAlias )
ELSE
  SETFLDSTATE( lcField, 3, lcAlias )
ENDIF
```

The first custom method, *SetValue(),* is called from the control's *Refresh()* method to update its value from the underlying data. This is normally handled automatically when you refresh bound controls but, since we have unbound the control behind the scenes, we have to write the code to handle it ourselves. We cannot allow empty values to reach the control, so we set its *Value* property to today's date if the *ControlSource* is either empty or null.

```
ltValue = EVALUATE( This.cControlSource )
IF NOT EMPTY( NVL( ltValue, '' ) )
  This.Object.Value = ltValue
ELSE
  This.Object.Value = DATETIME()
ENDIF
```

The second custom method, *UpdateControlSource(),* is called from the control's *Change()* method and, as the name implies, updates the underlying data from the control's *Value*. Normally, in most garden-variety controls (textboxes, combo boxes, and so on), this is handled automatically when the *Valid()* executes. However, OLE Containers do not have a *Valid()* method, so we had to find another solution. The *Change()* method fires whenever the control's *Value* changes, so it seems like a good place to update its *cControlSource.* Keep in mind that this code will fail if the *ControlSource* is a property instead of a field in a cursor.

```
WITH This
  REPLACE ( JUSTEXT( .cControlSource ) ) ;
    WITH ( .Object.Value ) IN ( JUSTSTEM( .cControlSource ) )
ENDWITH
```

To use the *acxDTPicker*, just drop it on a form and set it's *ControlSource*. The only code that you must write is to handle keystrokes for any *CallBack* fields specified in *CustomFormat*. You must also be aware that once you have set a date using this control, there is no way to reset it to an empty date. If you need to do this, you will either have to set its *CheckBox* property to True or add a separate checkbox to your form that states explicitly "Reset Date" and has code to blank the date in the underlying data.

How do I use the MonthView? (*Example: CH09.vcx::acxMonthView and MonthView.scx*)

The MonthView control is much more limited than either the Date and Time Picker or the Calendar controls. It looks like the calendar portion of the Date and Time Picker with no easy way to navigate to new months or years because you can only change one month at a time by clicking on the arrow (see **Figure 8**). One nice feature that it *does* have is that it allows you to

select a range of dates using a single control. In our opinion, this is the only time the control is useful. But it is only useful if it does not require the user to navigate to a month that is not in close proximity to the one displayed when the control is instantiated. The other downside is that, although the keyboard can be used to select an individual date in the control, we could not figure out how to select a range of dates without using the mouse!

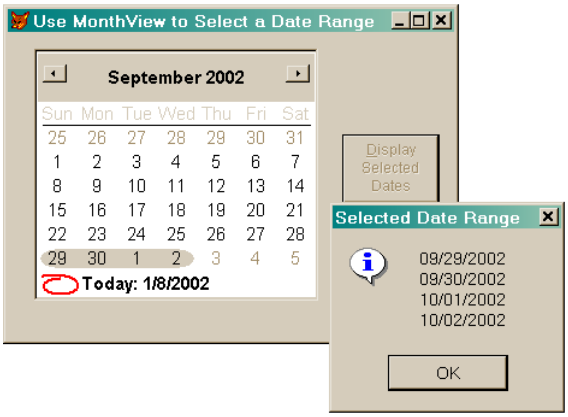


***Figure 8**. The MonthView control in action.*

There are only two properties that you need to be concerned with when using this control to select a date range: *SelStart* and *SelEnd*. These properties, as you would expect, contain the beginning and ending dates of the selected range. The dates are in DateTime format, so if you need to access all of the dates in the range, as the `MESSAGEBOX()` in the sample form does, you need to convert the beginning of the range to a Date value before manipulating it like this:

```
*** Display all the dates in the selected range
lcDisplayString = ''
*** Convert the datetime value to a date
ldDate = TTOD( Thisform.oMonthView.Object.SelStart )
DO WHILE ldDate <= TTOD( Thisform.oMonthView.Object.SelEnd )
  lcDisplayString = lcDisplayString + DTOC( ldDate ) + CHR( 13 )
  ldDate = ldDate + 1
ENDDO
MESSAGEBOX( lcDisplayString, 64, 'Selected Date Range' )
```

How do I use the ImageList?

The ImageList control is one that you will not use alone. Instead, you will use it in conjunction with other ActiveX controls that display images like the TreeView and ListView controls. The ImageList stores the images that are displayed by the control with which it is associated. You must populate the ImageList with all the required images before you bind it to another control (like a TreeView) because once you have done this, you can no longer delete images nor can you insert new images into the middle of its *ListImages* collection. You can, however, still add images to the end of the collection.

How do I store images in the ImageList?

You can store images in the control at design time using the ImageList properties or at run time by using the *Add()* method of its *ListImages* collection. If you are adding images visually, make sure that you set the size for the images on the General tab of its property sheet *before* you start to add them because once you have added images to the control, you cannot change the image size. Once you have done this, you are ready to add the images. Clicking on the Insert Picture button brings up a dialog that allows you to select an image (see **Figure 9**).

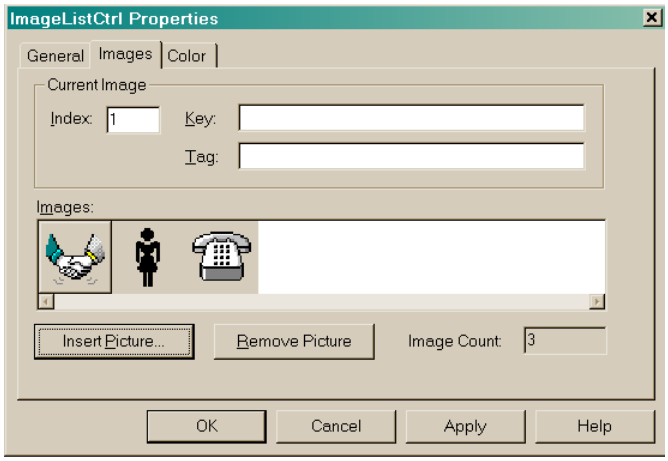


***Figure 9**. Adding images to the imageList at design time.*

To add images to the ImageList programmatically, you can use code like this:

```
This.ListImages.Add( [ nIndex ], [ cKey ], oPicture )
```

The *nIndex* argument is optional and specifies the image's order in the *ListImages* collection. If no index is specified, the image is added to the end of the list. The *cKey* argument is also optional and specifies a unique value used to identify the image, analogous to a primary key. The *oPicture* argument is required and must contain an object reference to a picture. This means that you will need to use the `LOADPICTURE()` function on the graphics file in order to add the image it contains to the control.

You can access the images in the *ListImages* collection using either the item's *Key* or its *Index*. As you will see later on in this chapter, you need the item's *Index* in order to assign an image to a *Node* in a TreeView or a ListItem in a ListView. However, it may not be possible to identify the correct image without using its *Key*. So it will not be unusual for you to use code like this to access your images after you have populated the control:

```
*** Find the image in the imageList
loImage = ThisForm.oImageList.Object.ListImages( <MyImageKey> )

*** And return the index
lnRetVal = loImage.Index
```

How do I bind the ImageList to other controls?

If you have examined the property sheet for the TreeView or ListView controls, you might believe that you can associate it with an ImageList at design time (see **Figure 10**).

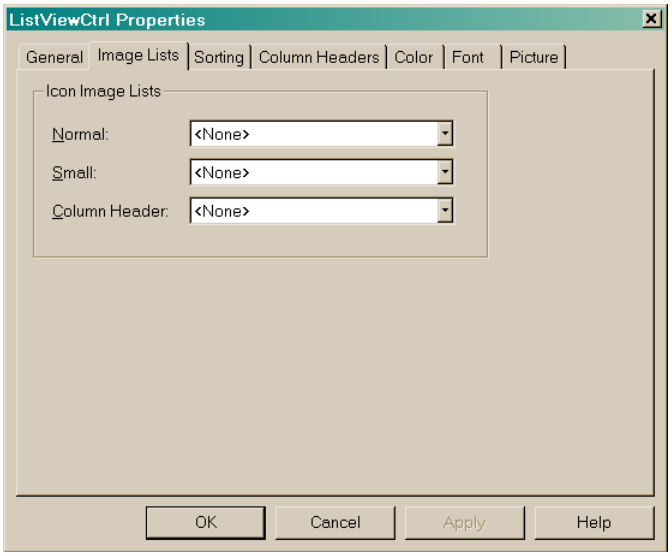


***Figure 10**. It looks like you can bind Image Lists at design time.*

Don't be fooled! This does not work. If you try to do this at design time, you will get the nasty OLE error in **Figure 11** at run time.

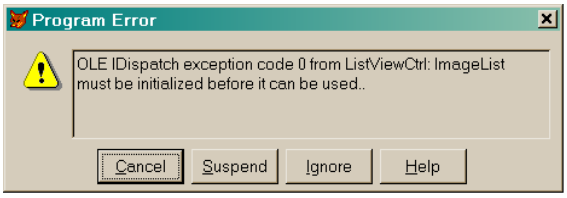


***Figure 11**. Results of binding Image Lists at design time.*

You must bind the ImageList to other controls using code in the *Init()* of your form. If you try to bind the ImageList in the *Init()* of the TreeView or ListView that needs it, you run the risk of throwing an error because the ImageList may not be instantiated yet. If you bind the ImageList in the Form's *Init()* you can be certain that both controls have been instantiated. This line of code is all it takes:

```
Thisform.oTree.ImageList = This.oImageList.Object
```

How do I use the ListView? (*Example: CH09.vcx::acxListView and ListView.scx*)

The first question really should be "When *should* I use a ListView?" The answer is whenever you want to display subsets of records that may be sorted in different ways. A ListView is a better choice than a grid for three reasons. First, because it allows the user to sort by any column just by clicking on the header. Second, it can display visual cues, so that it is always clear which column is controlling the sort and how it is sorted. Third, it allows you to assign different icons to each row that is displayed. All of these are possible with a native VFP grid (indeed, we showed how to do most of them in KiloFox), but it is hard work. Of course, if you must display large volumes of data, the native grid is still the better choice because of the overhead imposed by populating the ListView.

Each piece of data represented by the ListView is stored as a *ListItem* in its *ListItems* collection. This concept should be familiar since native Visual FoxPro ListBoxes also store their data in a *ListItems* collection (for more information on native Visual FoxPro ListBoxes, please refer to Chapter 5 in KiloFox). Each *ListItem* in a ListView consists of text and the index of an associated icon from an ImageList object (that is bound to the ListView). How the *ListItems* are actually displayed depends upon the setting of the ListView's *View* property.

When the ListView is displayed in "Icon" view (ListView.View = 0-lvwIcon), the data is displayed as shown in **Figure 12**.

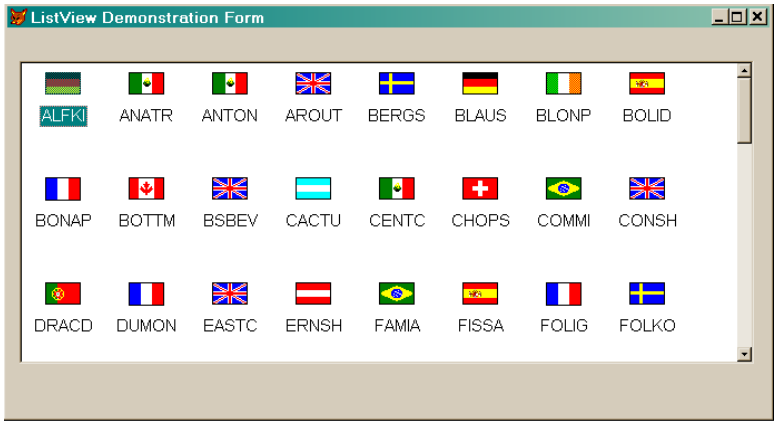


***Figure 12**. ListView displayed in icon format.*

If we switch to "Small Icon" view (*View = 1-lvwSmallIcon*), the same data appears as shown in **Figure 13**. Notice that the *ListItems* are displayed in order by row.

Is this beginning to look familiar? (Hint: Consider the different views that you can set in Windows Explorer.) Changing the *View* property to 2-lvwList results in a ListView that looks like **Figure 14**. Notice that in this view the *ListItems* are ordered by column, not row.

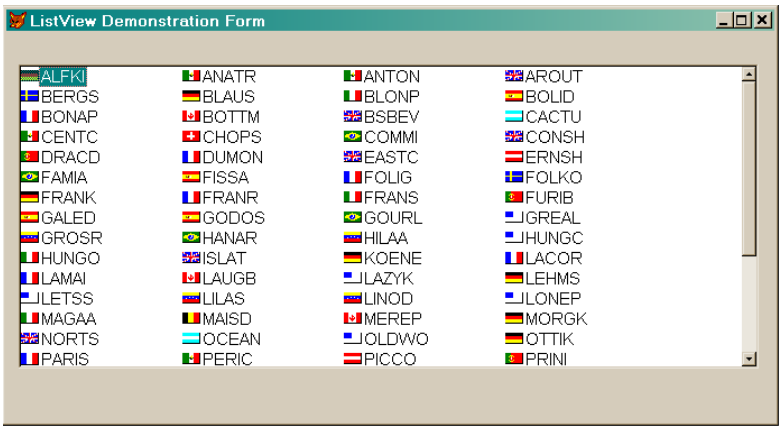


***Figure 13**. ListView displayed in small icon format.*

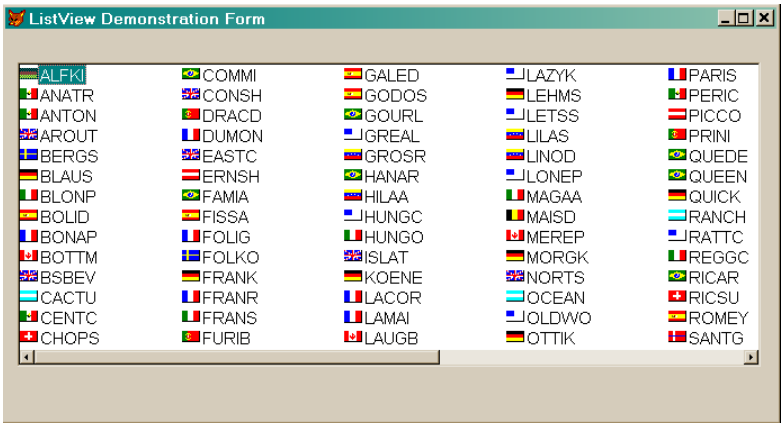


***Figure 14**. ListView displayed in list format.*

In our opinion, the most useful format is the fourth, the "Report" view (3-lvwReport), because it can display additional details for each *ListItem* (see **Figure 15**). In this view, which is column-based, the *ListItem* itself is always displayed in the first column. Each *ListItem* has a collection associated with it, named *SubItems*. Each element of data for the *ListItem* is stored as a *SubItem* in the *SubItems* collection, and each *SubItem* is represented by a column in the ListView.

When in Report view, the contents of the control can be sorted by any column, by clicking on the appropriate header. Very little code is required to implement this functionality and so our custom subclass uses the report view by default. The remainder of this section is based on the assumption that the ListView has been formatted this way.

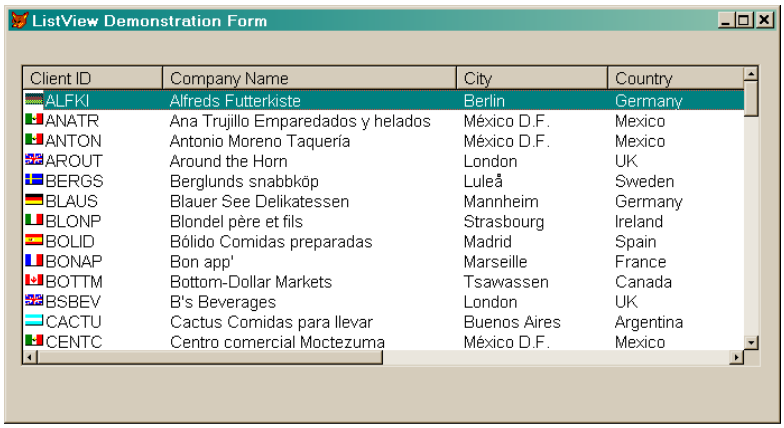


***Figure 15**. ListView displayed in report format.*

How do I add items to my ListView?

In all but Report view, all that is necessary is to call the *Add()* method of the *ListItems* collection to create one *ListItem* for each row of the source data.

```
loItem = oListView.ListItems.Add( Index, Key, Text, lnIcon, lnIcon )
```

The two *Icon* arguments are references to a "*Standard*" icon and a "*Small*" icon that are stored in two separate ImageLists. As discussed earlier, these ImageLists are bound to the ListView by setting the ListView's *Icons* and *SmallIcons* properties in the form's *Init()*:

```
Thisform.oListView.Icons = Thisform.oIcons
Thisform.oListView.SmallIcons = Thisform.oSmallIcons
```

For the Report view, yet another collection, named *ColumnHeaders*, is used to define the columns to be displayed. This can be defined visually in the properties sheet, or in code using its *Add()* method, like this:

```
oListView.ColumnHeaders.Add( index, key, text, width, alignment, icon)
```

where:

- *Index* is a unique integer that specifies the *ColumnHeader*'s order in the collection.
- *Key* is a unique character string (analogous to a primary key) that can be used to access the *ColumnHeader*.
- *Text* specifies the text that will appear in the *ColumnHeader*.
- *Width* is the width of the header in pixels.
- *Alignment* is either 0-Left, 1-Right, or 2-Center.

- *Icon* contains either the *index* or the *key* of the icon to be displayed in the header. This icon comes from an ImageList that is referenced by the ListView's *ColumnHeaderIcons* property.

All of these arguments are optional. Invoking the *ColumnHeaders Add()* method without specifying an *Index* adds the new *ColumnHeader* at the end of the existing collection. The first column defined (*Index* = 1) is reserved for the *ListItem*. Another collection (named *SubItems*) is created for each *ListItem* as *it* is defined, with one *SubItem* for each column having an index greater than one.

As a consequence the *SubItems* collection has no need of an *Add()* method of its own and, for a given *ListItem*, is populated like this:

```
WITH loItem
  FOR lnItem = 2 TO Thisform.oListView.ColumnHeaders.Count
    .SubItems( lnItem - 1 ) = "SubItem" + TRANSFORM( lnItem - 1 )
  ENDFOR
ENDWITH
```

How do I sort the items in my ListView?

First of all, the only way you can dynamically sort the items is when the ListView is in Report view. The ability to sort is controlled by the ListView's *Sorted* property. When set to True, a mouse click on any *ColumnHeader* fires the ListView's *ColumnClick* event. The actual sorting is handled by custom code in the method associated with that event.

The information about how the ListView is sorted is stored in two properties:

- *SortKey*: A zero-based value that specifies the controlling column. Note that because it is zero-based, it is always one less than the column index.
- *SortOrder*: An integer value defining the sort direction. Allowable values are 0-lvwAscending and 1-lvwDescending.

The *ColumnClick()* method takes as its single parameter an object reference to the *ColumnHeader* that was clicked. We need to examine the *SubItemIndex* property of this object so that we can make some decisions about what to do.

- If the *ColumnHeader*'s *SubItemIndex* is the same as the ListView's *SortKey*, it means that we have clicked on the column that is currently controlling the sort. If this is the case, we want to toggle the *SortOrder*.
- If the *ColumnHeader*'s *SubItemIndex* is different from the ListView's *SortKey*, it means that we have clicked on a column that is not controlling the sort. If this is the case, we want to set the current column as the controlling column. We do this by setting the ListView's *SortKey* to the current column's *SubItemIndex*.

A secondary issue involves controlling the visual cues. In order to supply these visual cues, we drop an ImageList onto the form and populate it with the icons that we want to be displayed in the headers (see **Figure 16**).

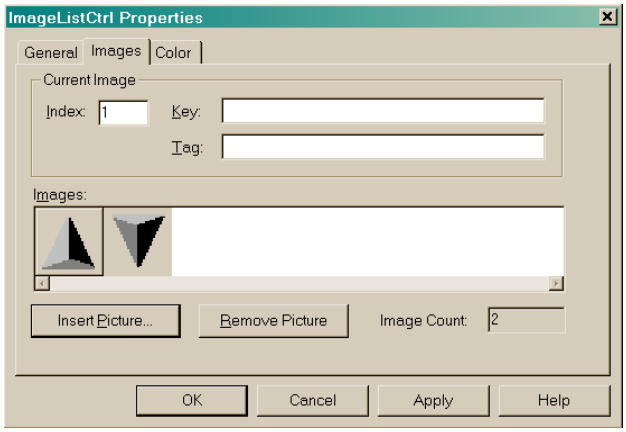


***Figure 16**. Icons to be used in the ColumnHeaders.*

Then, in the form's *Init()*, we bind the ImageList to the ListView like this:

```
ThisForm.oListView.ColumnHeaderIcons = ThisForm.oHeaderIcons
```

When we handle the sorting of the ListView, we must also handle updating the display of the icons in the *ColumnHeaders*. This is done by setting the *Icon* property of the *ColumnHeader* object to the *Index* of the appropriate icon in the ImageList or to zero in order to remove it.

So our code to handle the sorting looks like this:

```
LPARAMETERS columnheader

IF This.SortKey = ColumnHeader.SubItemIndex
  *** Toggle the sort order
  This.SortOrder = IIF( This.SortOrder = lvwAscending, ;
                        lvwDescending, lvwAscending )

  *** Display the correct icon
  ColumnHeader.Icon = IIF( This.SortOrder = lvwAscending, 1, 2 )
ELSE
  *** user has clicked on new column - initial sort order will become ascending
  *** Remove the icon from the column we were previously sorting by
  This.ColumnHeaders( This.SortKey + 1 ).Icon = 0
  This.SortOrder = lvwAscending
  This.SortKey = ColumnHeader.SubItemIndex
  ColumnHeader.Icon = 1
ENDIF
```

How do I know which item is selected?

As usual, the answer to this one is "It depends". The ListView control, much like the native Visual FoxPro ListBox, can contain more than one selected item if its *MultiSelect* property is set to True.

The ListView does not have a *ListIndex* property like its VFP brother, but it does have a *SelectedItem* property. This contains an object reference to the selected *ListItem* unless, of

course, this is a *MultiSelect* ListView. In this case, the ListView's *SelectedItem* always references the **first** *SelectedItem.* We can use the *SelectedItem* to get the information we need about the selected *ListItem* like its *Key* or its *Index*.

Related to the ListView's *SelectedItem* property is the *Selected* property of its *ListItems*. When a *ListItem* is *Selected*, this property is set to True. But be aware that setting the *Selected* property of a *ListItem* to True does not set the ListView's *SelectedItem* property, and thus does not cause the object to be selected. You can only use this property to determine whether a *ListItem* has already been selected by other means. Use code like this to find all of the selected items in a *MultiSelect* ListView:

```
LOCAL ARRAY laSelected
lnLen = 0
FOR EACH loItem IN ThisForm.oListView.ListItems
  IF loItem.Selected
    lnLen = lnLen + 1
    DIMENSION laSelected[ lnLen ]
    laSelected[ lnLen ] = loItem.Key
  ENDIF
ENDFOR
```

Can I make the ListView behave like a data-bound control?

The answer is an unqualified yes, and the class included with the sample code for this section is living proof of this fact. Our custom ListView class automagically synchronizes the data in the underlying cursor with the *SelectedItem* in the ListView. You do not need to write any additional code to find the record when the user clicks on any *ListItem* in the ListView, The key to making this approach work is to assign *Key* values to our *ListItems* when we create them that will enable us to easily locate the associated record in the underlying data. Since all of our tables always have a primary key field, this makes it easy. All we have to do is append the value of the primary key to the name of the cursor, separating the cursor name from the key value with an underscore, and use this as the *Key* for each *ListItem*.

The custom *acxListView* class has five custom properties that enable us to populate the control at runtime (see **Table 6**). All that is required is to ensure that the cursor specified as the data source is open.

***Table 6**. acxListView custom properties.*

Property	Description
cAlias	Name of the cursor that will be used to populate the ListView.
cPkField	Name of the primary key field in the cursor specified by *cAlias.*
cFkField	Name of the foreign key field that relates the *cAlias* to some parent cursor if the ListView should behave like a parameterized view.
uFkValue	Value of the foreign key to use to limit the contents of the ListView. When specified, only records with foreign key values that match this property are displayed in the ListView.
aDisplayFields	Two-dimensional array property that contains the field names and captions to be displayed from *cAlias*. If this is left empty, all fields from *cAlias* are displayed with the exception of the fields specified in the *cFkField* and *cPkField* properties. If the *cAlias* is a table in a DBC, `DBGETPROP()` is used to get the captions from the DBC. Otherwise, the field names are used as the captions.

To use the *acxListView* class, just drop it on a form along with three ImageLists. The ImageLists are required to set the ListView's *HeaderIcons*, *Icons*, and *SmallIcons* properties. Code like this in the form's *Init()* sets the ListView up:

```
LOCAL llretVal
llRetVal = DODEFAULT()
IF llRetVal
  *** Populate the listview
  WITH This.oList

    *** Set up the icons to indicate Sort Order
    *** when you click on the column header
    .ColumnHeaderIcons = ThisForm.oHeaderIcons

    *** And set up the icons for the list items
    .Icons = Thisform.oIcons
    .SmallIcons = Thisform.oSmallIcons

    *** Now call the methods to populate the list
    .CreateHeaders()
    .PopulateList()
    *** Locate the first record in the underlying data
    .FindRec( .SelectedItem )
  ENDWITH
  _VFP.AutoYield = .F.
  SYS( 2333, 0 )
ENDIF

RETURN llretVal
```

The reason that we call the ListView's custom *CreateHeaders()* and *PopulateList()* methods directly from the form is that, if we want to use the control's *aDisplayFields* property to display only certain fields, we need to set it up in the *Init()* of either the ListView or the form. The array must be defined before we attempt to use it to populate the ListView. We also need access to the icons in the ImageLists that are bound to the ListView in order to add the *ListItems*. As we discussed in the section on the ImageList, the best place to bind it to the ListView is in the form's *Init()* so that we can be certain that both controls are instantiated before we attempt to associate them. Therefore, it just makes good sense to perform all of the initial setup from the form's *Init()*.

The only other code that must be written is in the custom *GetIcon()* method of the instance of the ListView on the form. This is because we do not know how each specific ListView is going to associate the icons in its ImageList with specific *ListItems*, so this code **must** go in the instance. This is what the code in the Listview's *GetIcon()* method of our sample form looks like:

```
LOCAL loImage, lnRetVal

*** Look up the icon in the country table
IF SEEK( UPPER( ALLTRIM( Clients.cCountry ) ), 'Country', 'cCountry' )

  *** Find the image in the imageList
  loImage = ThisForm.oIcons.Object.ListImages( ALLTRIM( Country.cCountry ) )
```

```
  *** And return the index
  lnRetVal = loImage.Index
Else
  lnRetVal = 1
ENDIF

RETURN lnretVal
```

The custom *CreateHeaders()* method, as its name implies, creates the headers for the ListView based on the contents of its *aDisplayFields* array. The first thing that it does is to determine whether the array has already been populated. If it hasn't, this code populates the array with all the fields from the underlying cursor except for the ones specified in its custom *cPkField* and *cFkField* properties.

```
IF EMPTY( This.aDisplayFields[ 1, 1 ] )
  lnFldCount = AFIELDS( laFields, lcCursor )
  lnIndex  = 0
  FOR lnCnt = 1 TO lnFldCount

    *** Skip the key fields
    IF UPPER( ALLTRIM( laFields[ lnCnt, 1 ] ) ) == ;
       UPPER( ALLTRIM( This.cPkField ) )
      LOOP
    ENDIF
    IF UPPER( ALLTRIM( laFields[ lnCnt, 1 ] ) ) == ;
       UPPER( ALLTRIM( This.cfkField ) )
      LOOP
    ENDIF
    lnIndex = lnIndex + 1
    DIMENSION This.aDisplayFields[ lnIndex, 2 ]
    This.aDisplayFields[ lnIndex, 1 ] = laFields[ lnCnt, 1 ]

    *** Now, if we have a caption available, use that for the header
    *** So check to see if the data source is in a dbc and retrieve the caption
    *** if it is, otherwise use the name of the field
    lcDbc = CURSORGETPROP( "Database", lcCursor )
    IF NOT EMPTY( lcDbc )

      *** Make sure it is the current database
      SET DATABASE TO ( lcDBC )

      *** Get the caption for this field
      lcCaption = DBGETPROP( lcCursor + '.' + laFields[ lnCnt, 1 ], ;
                             'Field', 'Caption' )
      IF EMPTY( lcCaption )
        lcCaption = laFields[ lnCnt, 1 ]
      ENDIF
      This.aDisplayFields[ lnIndex, 2 ] = lcCaption
    ELSE
      This.aDisplayFields[ lnIndex, 2 ] = laFields[ lnCnt, 1 ]
    ENDIF
  ENDFOR
ENDIF
```

After we are certain that the control's custom *aDisplayFields* array is populated, we are ready to use the information to create the *ColumnHeaders*. Our class assumes a font of 9 point

Arial to calculate the *Width* for each *ColumnHeader*. It calculates the length of the header's caption as well as the length of the data in the underlying field and uses whichever is greater for the column width.

```
lnColumnCount = ALEN( This.aDisplayFields, 1 )
WITH This.ColumnHeaders
  FOR lnItem = 1 TO ALEN( This.aDisplayFields, 1 )

    *** Calculate the width for this column
    lnHdrWidth = ( LEN( This.aDisplayFields[ lnItem, 2 ] ) + 16 ) ;
                         * FONTMETRIC( 6, 'Arial', 9 )
    lnColWidth = LEN( TRANSFORM( EVALUATE( lcCursor + '. ' + ;
      This.aDisplayFields[ lnItem, 1 ] ) ) ) * FONTMETRIC( 6, 'Arial', 9 )
    IF lnColWidth < lnHdrWidth
      lnColWidth = lnHdrWidth
    ENDIF
    .Add( , , This.aDisplayFields[ lnItem, 2 ], lnColWidth,  lvwColumnLeft )
  ENDFOR
ENDWITH
```

Once the *ColumnHeaders* have been created, we are ready to add the *ListItems*. This code, in the custom *PopulateList()* method, scans the cursor specified in the ListView's custom *cAlias* property and calls the custom *CreateListItem()* method to create each *ListItem*. Notice that if a foreign key field has been specified, only records that match the specified value are used to populate the ListView.

```
lcCursor = This.cAlias
lcfk = This.cFkField
WITH This.ListItems

  *** Now we are ready to scan the cursor and add the list items
  SELECT ( lcCursor )

  *** Scan the entire thing if no FK field specified
  *** Otherwise, only scan for the appropriate records
  IF EMPTY( lcFk )
    SCAN
      This.Createlistitem()
    ENDSCAN
  ELSE
    SCAN FOR &lcFk = This.uFkValue
      This.Createlistitem()
    ENDSCAN
  ENDIF
ENDWITH
```

The custom *CreateListItem()* method uses the primary key in the current record to create the *ListItem* and assign it a *Key* value that can be used to uniquely identify it. This method also uses the fields specified in the *aDisplayFields* array to set up the *SubItems* for the current *ListItem*.

```
lcCursor = This.cAlias
WITH This.ListItems
  lnIcon = This.getIcon()
```

```
  *** Add the ListItem for this Record
  *** And set the ListItem's Key to the Name of the cursor and
  *** value of the PK field in the data source
  IF VARTYPE( lnIcon ) = 'N' AND NOT EMPTY( lnIcon )
    loItem = .Add( , lcCursor + '_' + ;
      ALLTRIM( TRANSFORM( EVALUATE( lcCursor + '.' + This.cPkField ) ) ), ;
      ALLTRIM( TRANSFORM( EVALUATE( lcCursor + '.' + ;
                This.aDisplayFields[ 1, 1 ] ) ) ), lnIcon, lnIcon )
  ELSE
    loItem = .Add( , lcCursor + '_' + ;
      ALLTRIM( TRANSFORM( EVALUATE( lcCursor + '.' + This.cPkField ) ) ), ;
      ALLTRIM( TRANSFORM( EVALUATE( lcCursor + '.' + ;
                This.aDisplayFields[ 1, 1 ] ) ) ) )
  ENDIF

  *** Use fields 2 - n to populate the list's SubItems
  WITH loItem
    FOR lnItem = 2 TO ALEN( This.aDisplayFields, 1 )
      .SubItems( lnItem - 1 ) = ALLTRIM( TRANSFORM( EVALUATE( ;
         lcCursor + '.' + This.aDisplayFields[ lnItem, 1 ] ) ) )
    ENDFOR
  ENDWITH
ENDWITH
```

The code that enables us to keep the ListView synchronized with the underlying data is in the custom *FindRec()* method. This method is called from the ListView's *ItemClick()* method, which fires whenever the user clicks on a *ListItem* or one of its *SubItems*. The *ItemClick()* method passes a reference to the *ListItem* that the user clicked on to *FindRec()*. The *FindRec()* method then uses the information in the *Key* of the passed *ListItem* to find the associated record in the cursor.

```
lnSelect = SELECT()

*** synchronize the underlying data source
lcKey = GETWORDNUM( toListItem.Key, 2, '_' )
lcPkField = This.cPkField
lcAlias = This.cAlias
lcType = TYPE(  lcAlias + '.' + lcPkField )
luVal = This.Str2Exp( lcKey, lcType )

*** Use seek if we have a tag
IF This.IsTag( lcPkField, lcAlias )
  llFound = SEEK( luVal, lcAlias, lcPkField )
ELSE
  *** must use locate
  SELECT ( lcAlias )
  LOCATE FOR &lcPkField = luVal
  llFound = FOUND()
  SELECT ( lnSelect )
ENDIF

RETURN llFound
```

You can see for yourself that the underlying data really does stay synchronized with the ListView by running the sample form and opening the DataSession window. Click on an item

in the ListView and then browse the Clients table. You will see that the record pointer is positioned on the record associated with the current *ListItem* in the ListView.

How do I use the ImageCombo? (*Example: acxImageCombo and ImageCombo.scx*)

The ImageCombo is useful in two specific situations. The first is when you want to display an icon for each item in the control's internal list—for example, if you wanted to display the flag alongside the name of the country in a dropdown list. This type of ImageCombo, which associates a specific icon with each item in the control's internal list, functions in a manner similar to our "data-bound" ListView described in the preceding section.

The second situation is when you want to display a hierarchical list. For example, if you need to display the contents of a directory that has subdirectories, the display is much clearer for the end user if the files in each subdirectory are indented. This type is unlikely to be data-driven because we need to construct the hierarchy for the entire data set that is to be displayed and this is best done in the instance.

Our ImageCombo subclass has six custom properties that allow it to simulate the behavior of a data-bound Visual FoxPro combo box when necessary (see **Table 7**). The ImageCombo demonstration form is shown in **Figure 17**.

***Table 7**. acxImageCombo custom properties.*

Property	Description
cAlias	Name of the cursor that will be used to populate the ImageCombo.
cPkField	Name of the primary key field in the cursor specified by *cAlias*.
cFkField	Name of the foreign key field that relates the *cAlias* to some parent cursor if the ImageCombo should behave like a parameterized view.
uFkValue	Value of the foreign key to use to limit the contents of the ImageCombo. When specified, only records with foreign key values that match this property are displayed in the list.
cDisplayField	Name of the field in *cAlias* to be displayed in the list.
cControlSource	Cursor.Field to update with the selections in the ImageCombo.

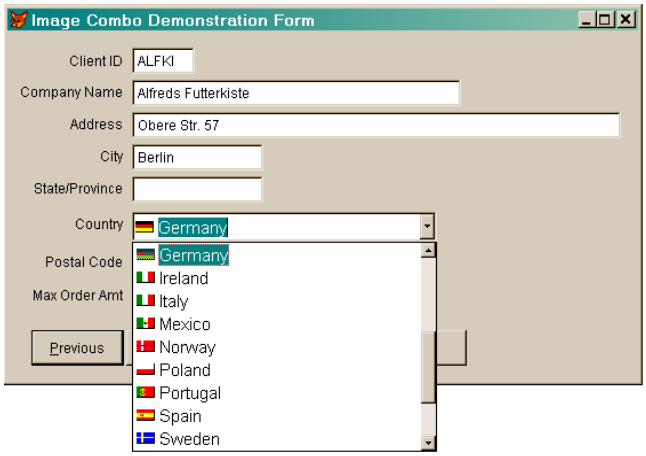


***Figure 17**. ImageCombo demonstration form.*

If you compare the custom properties for the ImageCombo to those listed in Table 6 for the ListView, you will notice that they are almost identical. There is a very good reason for this. Just as native Visual FoxPro list boxes are related to their combo box cousins, so are the ListView and the ImageCombo.

The ImageCombo has a *ComboItems* collection that you manipulate just the same way as the *ListItems* collection of the ListView. You add items to the ImageCombo's internal list by using the *Add()* method of its *ComboItems* collection like this:

```
loItem = Thisform.oImageCombo.ComboItems.Add( ;
           Index, Key, Text, Image, SelImage, Indentation )
```

The *Add()* method returns a reference to the newly created item. Like the ListView, the ImageCombo also has a *SelectedItem* property that holds the object reference to the selected item in the list. Much of the code in this class will look very familiar since it is so similar to the ListView.

When we were developing our ImageCombo class, we ran into several anomalies that are worth noting here. The first one is that, although the ImageCombo has a *ControlSource* property, you cannot use it! We discovered that we could bind the control directly to a field in a cursor and the form instantiated just fine. However, whenever we attempted to access the ImageCombo, it caused a C000005 error and crashed FoxPro! We could not even unbind the control behind the scenes once its *ControlSource* was set. We considered adding a custom *cControlSource* property to our class that could be used to bind it, but we quickly realized that it was almost impossible to write generic code to deal with it, and abandoned the idea in favor of instance-level code.

Since we could not bind our ImageCombo directly to a cursor, we had to write some code to handle updating the control's *SelectedItem* to reflect the state of the underlying data whenever it was refreshed. This was not a big problem. We created a custom template method called *SetValue()* and added some code to the class so that it was called from the control's *Refresh()* method. Then, in the form, we added the instance-specific code that was needed directly to the *SetValue()* method like this:

```
IF SEEK( UPPER( ALLTRIM( Clients.cCountry ) ), 'Country', 'cCountry' )
  lcKey = 'COUNTRY_' + TRANSFORM( Country.iCountryPK )
  This.SelectedItem = This.ComboItems( lcKey )
ELSE
  *** Set it to unknown
  *** We know that it is the first one in the table
  IF SEEK( 1, 'Country', 'iCountryPK' )
    This.SelectedItem = This.ComboItems( 'COUNTRY_1' )
  ENDIF
ENDIF
```

The more difficult task was updating the underlying data from the *SelectedItem* in the ComboList when a change was made. We added a custom template method called *UpdateControlSource()* to accomplish this. What was difficult was deciding where to call this method. The ImageCombo has a *Change()* method that seemed the obvious choice. However, it did not take us long to discover that we were mistaken. The ImageCombo's *Change* event does not fire when the user selects a new item from the list. Since the control does not have a

Valid() method, we finally had to settle for calling it from *LostFocus()*. This is the code in the instance that updates the data:

```
*** Find the data in the combo's "RowSource"
IF This.FindRec()
  *** and use it to update the cControlSource
  REPLACE cCountry WITH Country.cCountry IN Clients
ENDIF
```

To use our custom *acxImageCombo*, just drop it on a form along with an ImageList that contains the necessary icons. Then set the ImageCombo's *cAlias* to the name of the cursor that will be used to populate its *ComboItems* collection. Set its *cDisplayField* to the name of the field that will supply the *Text* of each *ComboItem*. Finally, set the *cPkField* properties to the name of the primary key field so the class can construct a *Key* for each *ComboItem* as it is added to the collection. You only need to supply a field name for the *cFkField* if you want to filter the ImageCombo on a value in some parent table.

The only code that must be written (other than the template methods discussed earlier), is this code called from the *Init()* of the form:

```
WITH This.oImgCombo
  *** And set up the icons for the list items
  .ImageList = Thisform.oImageList
  .PopulateList()
ENDWITH
```

and the ImageCombo's custom *GetIcon()* method that does exactly the same thing for the ImageCombo as the ListView's *GetIcon()* method does for the ListView.

How do I display a hierarchical list in the ImageCombo? (*Example: ImageComboTree.scx*)

When you examine the property sheet for the ImageCombo, you will see a property called *Indentation*. The Help file says that this property gets or sets the width of the indentation of objects in a control. It goes on to say that each indentation level is 10 pixels. However, we were unable to set this property to any value, either in the property sheet or in code, which has any effect on the width of the *Indentation* at any level!

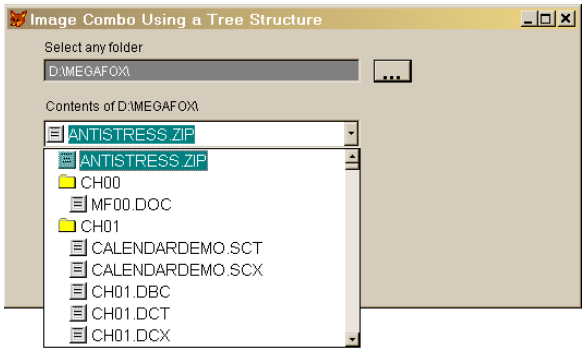


***Figure 18**. Using the ImageCombo to display a hierarchical list.*

All of the code required to supply the functionality resides in the sample form (see **Figure 18**). All we did was drop an OLE control on the form and insert the ImageCombo. The interesting code is in the form's custom *PopulateCombo()* method. It takes, as its parameters, the name of a directory and an indentation level. It then uses the **ADIR()** function to get an array of all the files and folders in the specified directory. The method loops through the array, processing each entry. When a directory is processed, it is added to the ImageCombo's *ComboItems* collection before the method calls itself recursively so that all of *its* files and subdirectories are included in the list.

```
lnDirCnt = ADIR( laDirs, tcDirectory + '*.*', 'D' )
IF lnDirCnt > 1
  FOR lnCnt = 1 TO lnDirCnt
    IF LEFT( laDirs[ lnCnt, 1 ], 1 ) # '.'
      IF DIRECTORY( ADDBS( tcDirectory + laDirs[ lnCnt, 1 ] ) )

        *** If we have a directory, add it to the ImageCombo
        Thisform.oImgCombo.ComboItems.Add( , , laDirs[ lnCnt, 1 ], ;
               1, 1, tnLevel )

        *** And then drill down to find all of its files
        Thisform.PopulateCombo( ADDBS( tcDirectory + laDirs[ lnCnt, 1 ] ),;
                                tnLevel + 1 )
      ELSE
```

If the current item in the array returned by **ADIR()** is either an icon or bmp, we decided to get a little fancy and use it as its own icon. We managed to do it, but we ran into some very interesting behavior when we first tried to get it to work. First of all, in order to associate an icon with a *ComboItem*, it must be present in the ImageList that is bound to the ImageCombo. So what we did was give each image in the ImageList a *Key* that was the same as its name. We mistakenly assumed that if we used code like this:

```
loImage = Thisform.oImageList.ListImages( JUSTSTEM( laDirs[ 1, 1 ] )
IF VARTYPE( loImage ) = 'O'
   *** The image is already in the ImageList
```

we could add our image to the list if it wasn't already there. Well, it didn't work. All it did was hand us back an OLE error. So we had to iterate through the entire *ListImages* collection to accomplish this. Even so, the code was still amazingly fast.

The other interesting behavior that we discovered was that if we used code like this:

```
FOR EACH loImage IN ThisForm.oImageList.ListImages
  IF loImage.Key = JUSTSTEM( laDirs[ 1, 1 ]
    .
    .
    .
  ENDIF
ENDFOR
```

the form refused to go away when we clicked on the Close button! Apparently this caused a dangling reference to the ImageList to persist even though all variables were declared as local. So the final code for adding the files to the ImageCombo looked like this:

```
*** If it is an icon or a bmp, let's see if it is in the imagelist
IF INLIST( UPPER( JUSTEXT( laDirs[ lnCnt, 1 ] ) ), 'ICO', 'BMP' )
  lcKey = JUSTSTEM( JUSTFNAME( laDirs[ lnCnt, 1 ] ) )
  llFound = .F.
  FOR lnIndex = 1 TO Thisform.oImgList.ListImages.Count
    IF Thisform.oImgList.ListImages( lnIndex ).Key = lcKey
      llFound = .T.
      EXIT
    ENDIF
  ENDFOR
  IF NOT llFound
    *** add the image to the image list and use it
    loImage = Thisform.oImgList.ListImages.Add(;
        , lcKey, LOADPICTURE( ADDBS( tcDirectory ) + laDirs[ lnCnt, 1 ] ) )
    lnIndex = loImage.Index
  ENDIF
ELSE
  lnIndex = 2
ENDIF
*** Just add the file name
Thisform.oImgCombo.ComboItems.Add( ;
           , , laDirs[ lnCnt, 1 ], lnIndex, lnIndex, tnLevel )
```

How do I use the TreeView? (*Example: CH09.vcx::acxTreeView and TreeView.scx*)

The TreeView is a good choice for displaying hierarchical data. It offers two specific advantages over using a series of related grids to display the same information. First, the relationships between the items are unmistakable when displayed in a TreeView. Second, when screen real estate is at a premium, the display requires a much smaller area without losing definition.

The basic principles for working with the TreeView are very similar to the ListView and ImageCombo, and this is not surprising when you consider that they all do much the same thing. While the ListView has a *ListItems* collection and the ImageCombo has a *ComboItems* collection, the TreeView has a *Nodes* collection. A *Node*, in this context, is simply the item of data that occupies a specific position in the hierarchy. Working with the TreeView requires two basic operations: adding *Nodes* and navigating between them.

How are Nodes added to the TreeView?

You add a *Node* to the TreeView using the *Add()* method of its *Nodes* collection but, because of the hierarchical nature of the TreeView, the syntax is a little different from that used to add items to the ListView and the ImageCombo. When you add a *Node*, you also need to specify the relationship that it has to other *Nodes* in the collection. So the syntax for adding a *Node* is:

```
oTree.Nodes.Add( relative, relationship, key, text, image, selectedimage)
```

where *Relative* is either the *Index* or the *Key* of an existing *Node*. The *Relationship* between the newly added *Node* and this existing *Node* can be:

- 0-tvwFirst: The new *Node* is positioned before all the *Nodes* at the same level of the hierarchy as the Node referenced by the *Relative* argument.

- 1-tvwLast: The new *Node* is positioned after all the *Nodes* at the same level of the hierarchy as the *Node* referenced by the *Relative* argument.
- 2-tvwNext: (Default) The new *Node* is positioned after the *Relative Node* as a sibling *Node.*
- 3-tvwPrevious: The new *Node* is positioned before the *Relative Node* as a sibling *Node.*
- 4-tvwChild: The new *Node* is added as a child to the *Relative*'s *Nodes* collection.

If no *Relative Node* is specified, the newly created *Node* is added at the end of the top level of the hierarchy. The remaining arguments should look quite familiar. They work exactly the same way here as they do when adding *ListItems* to the ListView or *ComboItems* to the ImageCombo. The *Add()* method returns an object reference to the newly created *Node*.

The main problem with adding *Nodes* to the TreeView control when it is instantiated is that it can take a relatively long time to populate the entire control. A more efficient approach is, therefore, to populate the top-level *Nodes* only and add child *Nodes* when the parent *Node* is expanded. The easiest way to implement this is to add a "dummy" *Node* to each top-level *Node* as it is created.

This is required if we want to be able to expand any of the top-level *Nodes* because the little "+" will not appear to the left of the *Node* unless it has at least one child *Node*. Then all we have to do is write a little code in the TreeView's *Expand()* method to add the child *Nodes* if the "dummy" *Node* still exists.

One interesting little quirk of the TreeView control that is worth noting here is that you must also set its *LineStyle* property to 1-Root Lines if you want to see the plus/minus signs displayed for top-level items.

How do I navigate the TreeView?

The TreeView control has a single *SelectedItem* property that contains an object reference to the currently selected *Node*. It is the *Nodes* collection and the individual *Node* objects that have the properties required to navigate the tree. You can access an individual *Node* in the collection using either its *Key* or its *Index* like this:

```
Thisform.oTree.Nodes( Key ) or Thisform.oTree.Nodes( Index )
```

However, the value of its *Index* is entirely dependent on the order in which the *Node* is added to the TreeView, so it is only useful in those instances where you need to iterate through all the children in the *Nodes* collection of a specific *Node*.

These properties of the *Node* object enable you to get information about other *Nodes* that are related to it.

- *Root* is an object reference to the *Node* that is displayed at the top of the TreeView. This is only the root *Node* of the currently selected *Node* if the TreeView contains one root-level *Node*.
- *Parent* is an object reference to the owner of the current *Node.*

- *Child* is an object reference to the first item owned by the current *Node*.
- *Children* is the number of items owned by the current *Node*.
- *FirstSibling* is an object reference to the first *Node* at the same level of the hierarchy as the current *Node*. The *Node* that is referenced by the *FirstSibling* property changes when new *Nodes* are added to this level of hierarchy and 0-tvwFirst is specified as the relationship.
- *LastSibling* is an object reference to the last *Node* at the same level of the hierarchy as the current *Node*. The *Node* that is referenced by the *LastSibling* property changes when new *Nodes* are added to this level of hierarchy and 1-tvwLast is specified as the relationship.
- *Next* is an object reference to the sibling *Node* that follows the current node.
- *Previous* is an object reference to the sibling *Node* that the current node follows.

How does the acxTreeView class work?

Our subclass of the TreeView control is data driven and is designed to be dropped onto a form and implemented with the minimum of instance-level code. Like our custom ListView class, it simulates the behavior of a data-bound control in that the underlying cursors automatically stay in synch with the selected *Node* in the TreeView.

How are the nodes managed?

Our TreeView class uses a five-column array, named *aLevels*, to store the details of the cursors used to populate the *Nodes* collection (see **Table 8**). Each row contains the information required to populate the collection for that level of the hierarchy. For example, the first row of the array contains information to populate all of the root level *Nodes* in the TreeView. The second row of the array contains information to populate all of the children of the root level *Nodes* and so on.

***Table 8**. Information held in acxTreeView.aLevels array property.*

Column	Description
1	Alias that supplies the data for this level of the hierarchy.
2	Name of primary key field in the alias contained in column 1.
3	Name of the foreign field that relates this item to its parent (unless this is a root-level *Node*).
4	Name of the field in the alias contained in column 1 that supplies the *Node*'s *Text*.
5	A logical flag to force a LOCATE (instead of a SEEK) when synchronizing the TreeView with the cursor specified in column 1.

So, to use the class in our sample form, all we had to do was add this code to the custom *SetForm()* method and call it from the form's *Init()* to populate the *aLevels* array (see **Figure 19**). Once this array is populated, all that is left to do is call the TreeView's custom *AddNodes()* method to create all the root-level *Nodes*.

```
WITH This.otree
  .ImageList = This.oList.Object
  _VFP.AutoYield = .F.
  SYS( 2333, 0 )
  DIMENSION .aLevels[ 3, 5 ]
  .aLevels[ 1, 1 ] = 'CLIENTS'
  .aLevels[ 1, 2 ] = 'ICLIENTPK'
  .aLevels[ 1, 4] = 'CCOMPANY'
  .aLevels[ 2, 1 ] = 'CONTACTS'
  .aLevels[ 2, 2 ] = 'ICONTACTPK'
  .aLevels[ 2, 3 ] = 'ICLIENTFK'
  .aLevels[ 2, 4] = 'ALLTRIM( CFIRST ) + [ ] + ALLTRIM( CLAST )'
  .aLevels[ 3, 1 ] = 'PHONES'
  .aLevels[ 3, 2 ] = 'IPHONEPK'
  .aLevels[ 3, 3 ] = 'ICONTACTFK'
  .aLevels[ 3, 4] = 'CNUMBER'

  *** Now load the level 1 Nodes of the treeview
  .AddNodes()

  *** Get the form set up for the first time
  GO TOP IN Clients
  Thisform.RefreshForm()
ENDWITH
```

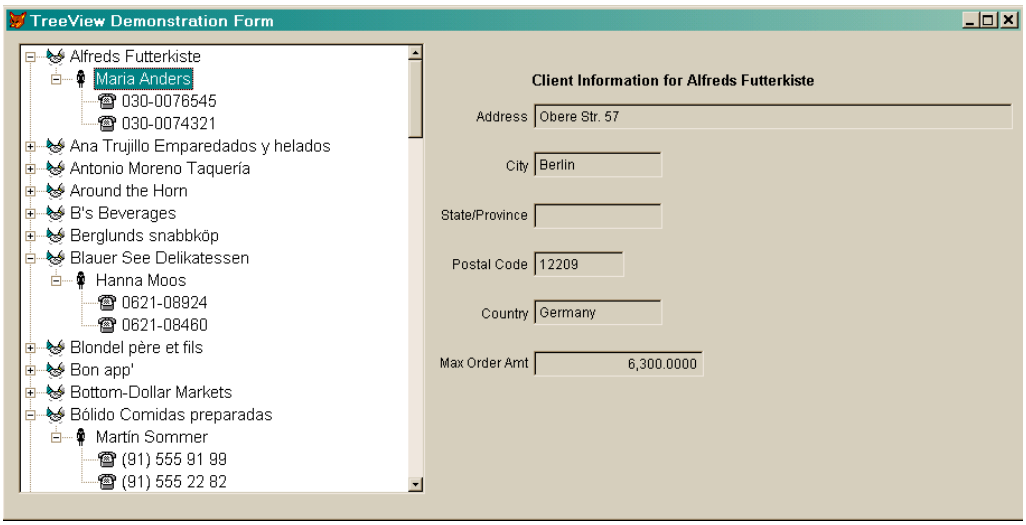


***Figure 19**. The TreeView control in action.*

The custom *AddNodes()* method is designed so that when called with no parameters, it creates the root-level nodes:

```
*** First see if we are populating level 1
IF EMPTY( tcNodeKey )
  lcAlias = This.aLevels[ 1, 1 ]
  lcKeyField = This.aLevels[ 1, 2 ]
  lcTextField = This.aLevels[ 1, 4 ]
```

```
  SELECT ( lcAlias )
  SCAN
    lcKey = UPPER( ALLTRIM( lcAlias ) ) + '_' + ;
               ALLTRIM( TRANSFORM( EVALUATE( lcKeyField ) ) )
    This.Nodes.Add( , tvwLast, lcKey, ;
                    ALLTRIM( TRANSFORM( EVALUATE( lcTextField ) ) ), 1)

    *** Now add dummy Nodes for all the branches
    This.Nodes.Add( lcKey, tvwChild, lcKey + '_DUMMY', 'DUMMY')
  ENDSCAN
```

When the user clicks on the plus sign, the *Expand()* method checks to see if the *Node* has already been expanded. If not, the *Node* will contain only a single `'DUMMY'` child *Node*. In this case we call the *AddNodes()* method and pass the *Key* of the *Node* being expanded:

```
ELSE
  *** Check to see if this Node is already populated.
  *** We do not want to populate it again if it is
  *** see if we need to populate child Nodes
  IF ( This.Nodes( tcNodeKey ).Children > 0 ) AND ;
     ( This.Nodes( tcNodeKey ).Child.Text = 'DUMMY' )
    This.Nodes.Remove( This.Nodes( tcNodeKey ).Child.Index )
```

After the `'DUMMY'` *Node* is removed, we use the passed *Key* to find the row in the *aLevels* array that contains the information for the *Node* that is being expanded. We can do this because the first word of the *Key* is the name of the alias that is associated with it. We just use `ASCAN()` to find the number of the row that contains this alias in its first column.

```
    lcParentAlias = GETWORDNUM( tcNodeKey, 1, '_' )
    lnLevel = ASCAN( This.aLevels, lcParentAlias, -1, -1, 1, 15 )
```

The name of key field in the cursor is in the second column of the array. We use the data type of this field to convert the second word in the passed *Key* value from character to the correct data type for use in scanning the child alias for all the related child records. The second word of a *Node*'s *Key* always contains the string representation of its underlying data's primary key.

The information for the child alias is located in the very next row of the array unless, of course, the *Node* we are trying to expand is a leaf *Node* (that is, a *Node* that has no children). If this is the case, we must be on the last row of the array.

```
    lcParentKeyField = This.aLevels[ lnLevel, 2 ]
    lcParentKey = GETWORDNUM( tcNodeKey, 2, '_' )
    lcType = TYPE( lcParentAlias + '.' + lcParentKeyField )
    luKeyVal = This.Str2Exp( lcParentKey, lcType )

    *** Now move to the row in the array that contains
    *** the information for the cursor that is used
    *** to create the child Nodes
    lnLevel = lnLevel + 1

    *** Make sure we are not trying to expand a leaf Node
    IF lnLevel <= ALEN( This.aLevels, 1 )
```

Finally, we get the name of the child alias along with the field names of its primary and foreign key fields as well as the field to be used for the child *Node*'s *Text* and scan the child table for only those records that match the primary key in the record associated with the parent *Node.* We use this information to add each child *Node* to the TreeView.

```
      lcAlias = This.aLevels[ lnLevel, 1 ]
      lcKeyField = This.aLevels[ lnLevel, 2 ]
      lcFKField = This.aLevels[ lnLevel, 3 ]
      lcTextField = This.aLevels[ lnLevel, 4 ]
      SELECT ( lcAlias )
      SCAN FOR EVALUATE( lcFKField ) = luKeyVal
        lcKey = UPPER( ALLTRIM( lcAlias ) ) + '_' + ;
                ALLTRIM( TRANSFORM( EVALUATE( lcKeyField ) ) ) + '_' + ;
                ALLTRIM( TRANSFORM( EVALUATE( lcFkField ) ) )
        This.Nodes.Add( tcNodeKey, tvwChild, lcKey, ;
                ALLTRIM( TRANSFORM( EVALUATE( lcTextField ) ) ), lnLevel )
        IF lnLevel < ALEN( This.aLevels, 1 )
          This.Nodes.Add( lcKey, tvwChild, lcKey + '_DUMMY', 'DUMMY')
        ENDIF
      ENDSCAN
    ENDIF
  ENDIF
ENDIF
```

Whenever the user clicks on a *Node* in the TreeView, its *NodeClick* event fires. It is worth mentioning that, even though a *Node* is supposed to be selected when it is clicked, it does not always happen. To ensure that the *Node* that is clicked becomes the TreeView's *SelectedItem*, include this line of code in the *NodeClick()* method:

```
Node.Selected = .T.
```

Having ensured that the clicked node is selected, we then call the class's custom *SynchCursors()* method to synchronize the cursors. This first parses out the required cursor alias and key from the node key and then walks up the hierarchy to find all of its parent records.

```
loNode = toNode
lnSelect = SELECT()

*** Parse out the alias and PK from the Node's key
*** All keys in the form Alias_PkValue_FkValue( where applicable )
lcAlias = GETWORDNUM( toNode.Key, 1, '_' )
lcPK = GETWORDNUM( toNode.Key, 2, '_' )
lcFK = GETWORDNUM( toNode.Key, 3, '_' )

*** Find the record in the alias associated with the currently selected Node
lnLevel = ASCAN( This.aLevels, lcAlias, -1, -1, 1, 15 )
IF This.FindRec(  lcPK, lnLevel )

  *** Go ahead and find the parent records all the way up the tree
  DO WHILE NOT ISNULL( loNode )
    loNode = This.GetParentNode( loNode.Key )
  ENDDO
```

The *GetParentNode()* method takes a single parameter, which is the key of a node. It returns either an object reference to the parent (and finds the record for the parent) or, if there is no parent node, a null value. Having found all the parent data, we then need to check for child data (a hierarchy can be traversed in two directions).

```
  *** And find the first record of any children all the way down
  *** to the last leaf on this branch if there are kids
  loNode = toNode
  DO WHILE NOT ISNULL( loNode )
    loNode = This.GetChildNode( loNode.Key )
  ENDDO
ENDIF
```

GetChildNode() takes one required parameter, which is the *Key* for a *Node* object. A second (optional) parameter is used to define the *Index* of the child *Node* to return. If no *Index* is passed, the Index defaults to 1. It returns either the object reference to the specified child node or, if no child node exists, a null value.

How are the context-sensitive menus managed?

The trick here is to determine exactly where the user has clicked. Since the TreeView control does not expose a *RightClick()* method, we must use its *HitTest()* method to determine whether there is a node under the mouse. The method returns an object reference to the *Node* located at the X and Y coordinates that are passed to it. If there is no *Node* at the specified coordinates, it returns NULL. The problem is that the *HitTest()* method expects to receive the coordinates in *Twips* but Visual FoxPro defines them in *Pixels*, so we have to convert between the two sets of units.

Two custom properties, *nFactorX* and *nFactorY*, are used to store the conversion factors, and they are set by calling the custom *SetFactors()* method from the TreeView's *Init()*.We would like to thank Doug Hennig for making this code available on his Web site so that we did not have to do all the hard work that he originally did.

```
LOCAL liHDC, liPixelsPerInchX, liPixelsPerInchY

#DEFINE PIXELS_X      88
#DEFINE PIXELS_Y      90
#DEFINE TWIPS_PER_INCH  1440

DECLARE INTEGER GetDC IN WIN32API INTEGER iHDC
DECLARE INTEGER GetDeviceCaps IN WIN32API INTEGER iHDC, INTEGER iIndex

liHDC  = GetDC( Thisform.HWnd )

*** Get the pixels per inch.
liPixelsPerInchX = GetDeviceCaps( liHDC, PIXELS_X )
liPixelsPerInchY = GetDeviceCaps( liHDC, PIXELS_Y )
```

```
*** Get the twips per pixel.
WITH THIS
  .nFactorX = TWIPS_PER_INCH / liPixelsPerInchX
  .nFactorY = TWIPS_PER_INCH / liPixelsPerInchY
ENDWITH
```

A third custom property of our TreeView class, *cMenu*, contains the name of a shortcut menu to display when the user right-clicks on a *Node*. We can use this code in the *MouseUp()* method to determine which mouse button was pressed and take appropriate action.

```
IF Button = 2 AND NOT EMPTY( This.cMenu )

  *** Get a reference to the Node under the mouse
  loNode = This.HitTest( X * This.nFactorX, Y * This.nFactorY )

  *** If we have a valid Node, show the menu
  IF NOT ISNULL( loNode )
    This.ShowMenu()
  ENDIF
ENDIF
```

One interesting behavior that we noticed was that right-clicking on either the icon or displayed text of a *Node* selects that *Node*. However, right-clicking the leading plus/minus sign does not.

Incidentally, another use for the *HitTest()* method would be to implement drag-and-drop functionality in the TreeView. However, because it is so application-specific, we have left that as an exercise for the reader.

And finally

This may, at first sight, look like a terribly complicated way to manage things. However, the benefit of this design is that all of the code to populate and manage the nodes is generic. The only code required to use this class is the setup code that populates the custom *aLevels* array for your data.

How do I synchronize a TreeView with a ListView? (*Example: TreeAndList.scx*)

This task is much easier than you think, especially if you use our custom acxTreeView and acxListView subclasses. All you need to do is drop them on a form along with any ImageLists to supply their icons and ensure that the cursors that will be used to populate the controls are open and available. Very little instance-level code is required to accomplish the task.

Our sample form displays all of the customers in the Tasmanian Traders sample application that ships with Visual FoxPro along with their orders in the TreeView on the left (see **Figure 20**). When a customer *Node* is selected in the TreeView, all of the orders for that customer are displayed in the ListView on the right. When an order *Node* is selected in the TreeView, all of its order lines are displayed in the ListView.

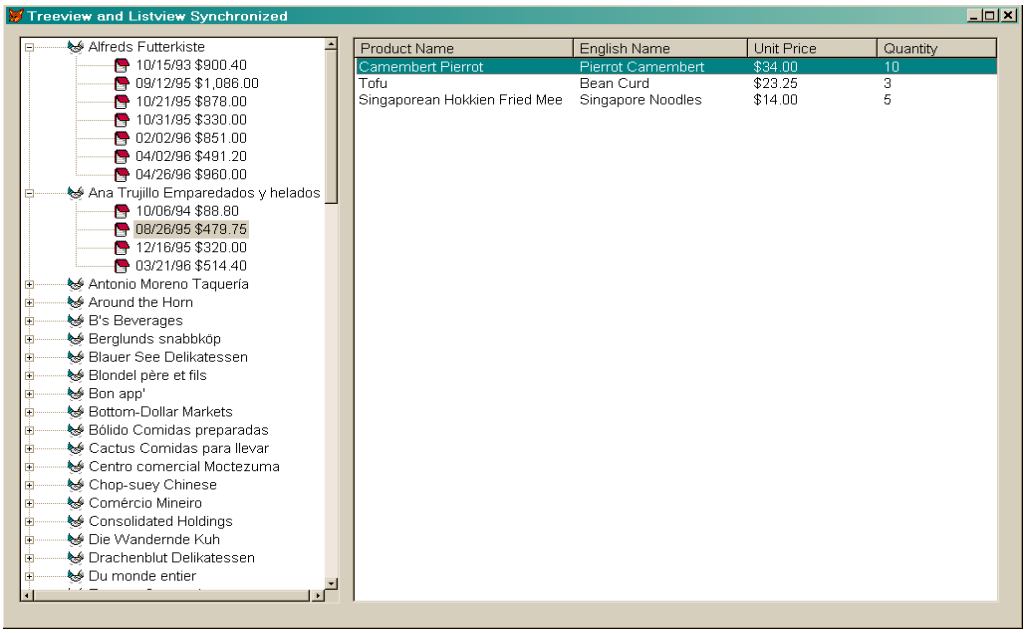


***Figure 20**. The TreeView control synchronized with a ListView.*

The key to making this work is the addition of two custom methods to the form. The first one, *SynchListView()*, is called from the TreeView's *NodeClick()* method and is passed the *Key* of the selected node in the TreeView. Since the first word of the *Node Key* is the alias of the cursor that was used to create the node, we can use this piece of information to determine if we are changing the data source for the ListView. After clearing the contents of the ListView, this method sets its properties so that it can re-populate itself with the items appropriate for the selected *Node* in the TreeView.

```
LPARAMETERS tcNodeKey
LOCAL lcParentAlias, lcCursor

Thisform.oListView.ListItems.Clear()

*** We are going to populate the listview with either orders
*** or line items depending on which node we have clicked on in the treeview
lcParentAlias = GETWORDNUM( tcNodeKey, 1, '_' )
IF lcParentAlias = 'CUSTOMER'
  lcCursor = 'ORDERS'
ELSE
  lcCursor = 'LV_ORDERLINES'
ENDIF

*** See if we are changing datasources for the list view
IF lcCursor # UPPER( ALLTRIM( Thisform.oListview.cAlias ) )
  *** We need to re-set the display fields and
  *** re-create the columns for the listview
  Thisform.SetListView( lcCursor )
ENDIF
```

The second custom method, *SetListView()*, is called here from the *SynchListView()* method as well as from the *SetForm()* method of the form when it is instantiated. This method first clears the contents of the ListView's *ColumnHeaders* collection. Then, depending on the name of the cursor it is passed, it sets the values for the ListView's custom *cAlias*, *cFkField*, and *cPkField* properties. Finally, it populates the control's *aDisplayFields* array before calling its *CreateHeaders()* method to populate the *ColumnHeaders* collection.

Once the ListView has the correct *ColumnHeaders* in place, the *SynchListView()* method ensures that the correct *ListItems* are generated for the ListView like this:

```
WITH Thisform.oListView
  *** Make sure we clear any icons from the header in the ListView
  IF .SortKey >= 0
    .ColumnHeaders( .SortKey + 1 ).Icon = 0
  ENDIF
  *** Now Populate the listview
  IF lcCursor = 'ORDERS'
    .uFkValue = ALLTRIM( Customer.Cust_ID )
  ELSE
    vp_Order_ID = Orders.Order_ID
    REQUERY( 'lvOrderLines' )
    .uFkValue = ''
  ENDIF
  .PopulateList()

  *** And make sure the underlying cursor is on the correct record
  .FindRec( .SelectedItem )
ENDWITH
```

This really is just about all that it takes to synchronize the two controls! The only reason that we can do this so easily is that our design enables these controls to populate and manage their internal collections merely by setting a few custom properties. We think that our approach has really paid off.

Controls for animation and sound

There are three ActiveX controls that are specifically concerned with handling animation and sound that can be used to extend your Visual FoxPro applications. Both the Animation control (MSCOMCT2.OCX) and the Multimedia MCI control (MCI32.OCX) ship with VFP and should therefore be fairly reliable. However, the functionality that they provide is very specific and quite limited. The Windows Media Player (MSDXM.OCX), which can be installed as part of the Windows setup, has more functionality, but is also a much more complex control.

How do I animate a form?

Figure 21 shows a Visual FoxPro version of the Windows "copying files" display. This little form uses the ActiveX Animation control (contained in MSCOMCT2.OCX, associated Help file CMCTL298.CHM) to run an AVI file asynchronously in the background while VFP carries on executing in the foreground.

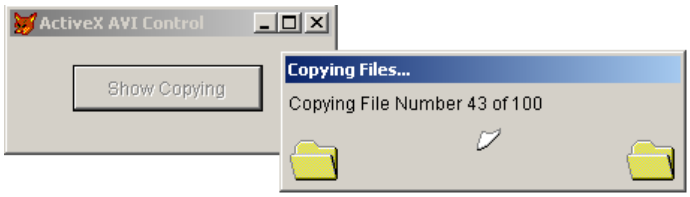


***Figure 21**. Animated "copying files" display.*

It is important to recognize that the Animation control is very limited in what it can do! In fact, it can play only Audio Video Interleaved (AVI) files that have no sound and that are either uncompressed or have been compressed using Run-Length Encoding (RLE). However, such files are easy to come by, and are easy to create using the appropriate software. This example uses one of the AVI files that ships with Windows and that (for convenience) is also included in the sample code for this chapter.

The "copying files" dialog class (*Example: CH09.vcx:: xCopyFile*)

As with the previous ActiveX controls, we have created our own subclass for the Animation control in CH09.VCX, named *xAviView*. The only changes to the default settings are that we have set the *AutoPlay* property to .T. and the *Background* property to transparent instead of opaque.

The dialog is also defined as a class, named *xCopyFile*, in CH09.VCX. This is a form class that has been set up to be always on top and auto-centering. Notice that it has no controls at all—not even the standard Close button, which has been removed by setting the *ControlBox* property to *False*. The only thing that caused any problems in defining this class was sizing the form so as to display the AVI correctly. We found no good way to do it other than trial and error.

Like the progress bar dialog described earlier in this chapter, we gave this class the ability to show a label in addition to the ActiveX control. As a glance at the class library will show, there is remarkably little code required. The *Init()* method accepts a single parameter, which it passes on to the custom *SetLabels()* method, and it then starts the animation by calling the *Open()* method of the animation control and passing the required file name (in this case, AVICOPY.AVI). Remember, this instance of the control is based on our own subclass in which we set the *AutoPlay* property to *True* so that we do not need to explicitly start the animation.

If you need finer control, so as to be able to stop and start an animation, leave the *AutoPlay* set to False. The *Open()* method now functions purely as a "loader" for the specified AVI file, and you can control the animation by calling the control's *Play()* and *Stop()* methods explicitly as needed.

Using the "copying files" dialog (*Example: frmAVICopy.scx*)

The sample form simply instantiates an instance of the dialog class and updates the label display inside a loop as follows:

```
LOCAL loCyFile, lnCnt
*** Create the file copy form, and set its caption
loCyFile = NEWOBJECT( 'xcopyfile','ch09ak.vcx' )
loCyFile.Visible = .T.
*** Update Comment while the animation runs
```

```
FOR lnCnt = 1 TO 100
  loCyFile.SetLabels("Copying File Number " + TRANSFORM( lnCnt ) + " of 100 ")
  INKEY(0.1,'h')
NEXT
RELEASE loCyFile
```

That is all that there is to using this control. It really is very simple and, for providing basic animation of the type illustrated here, is perfectly satisfactory. It should also be noted that one apparent shortcoming of this control is that it does not seem to respect Visual FoxPro's **SET PATH** command. When using the control's *Open()* method, if the specified .avi file is not in the current directory, the fully qualified path name of the file must be specified.

How do I add sound to my application?

There are several possible ways of adding sound to a VFP application. The first is to use the native Visual FoxPro **SET BELL** and **?? CHR(7)** commands. The set bell command is used to define a WAV file and then the in-line print command uses ASCII Code 7 to play it, as follows:

```
SET BELL TO "RINGIN.WAV"
?? CHR(7)
```

This works very nicely for simple sounds (such as an audible alert), but you do not have any control over how the sound is reproduced and, more importantly, the sound is always played synchronously so that no other activity is possible in VFP while the sound file is playing. This is not an issue when all that is required is a simple "*ding*" but could be a problem if you needed to play a sound that lasts for more than a few milliseconds.

If you need absolute control over sound (or any other supported media, for that matter), you can access the Windows Media Control Interface (MCI) application programming interface and call the necessary functions directly from your code. (For details on how to access the various Windows APIs, see Chapter 10.) However, unless your requirements are very specialized, there is a third option: Use the MultiMedia ActiveX control (contained in: MCI32.OCX, associated Help file MMEDIA98.CHM). As the file name suggests, this control is actually a wrapper for the MCI functions and is capable of much more than simply reproducing an existing WAV file.

Subclassing the multimedia control (*Example: CH09.vcx:: xMMedia*)

By default the MultiMedia control provides access to MCI functions through a user interface that appears as series of VCR-style buttons (see **Figure 22**). However, the control's property sheet exposes two properties for each button, which determines whether the functionality controlled by that button is available, and whether the button itself is visible.

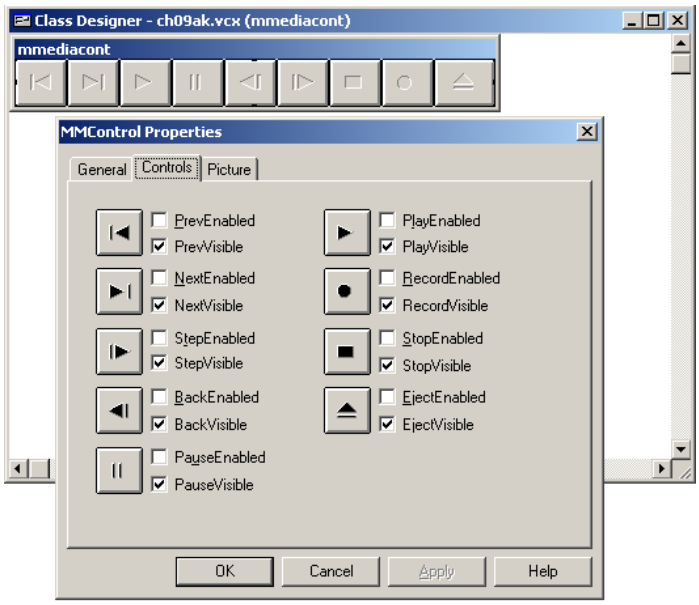


***Figure 22**. The user interface for the MultiMedia control.*

As implied by the name, the MultiMedia control is capable of dealing with much more than simply playing sound files. In fact, it is capable of interacting with a variety of different devices, and its behavior is actually controlled by the setting of the *DeviceType* property. The supported devices and their associated values for the *DeviceType* property are listed in **Table 9**.

***Table 9**. Devices supported by the MultiMedia control.*

Device	DeviceType	File	Description
CD audio	CDAudio		CD audio player
Audio Tape	DAT		Digital audio tape player
Video	DigitalVideo		Digital video in a window (not GDI-based)
Other	Other		Undefined MCI device
Overlay	Overlay		Overlay device
Scanner	Scanner		Image scanner
Sequencer	Sequencer	mid	Musical Instrument Digital Interface (MIDI) sequencer
Vcr	VCR		Video cassette recorder or player
AVI	AVIVideo	avi	Audio Visual Interleaved video
Videodisc	Videodisc		Videodisc player
Wave audio	Waveaudio	wav	Wave device that plays digitized waveform files

Closely associated with the *DeviceType()* is the *FileName* property, which will either be the fully qualified path and name of a file to be played or the path to the specified device (for instance, the drive letter for a CD player). These properties are exposed on the first page of the control's property sheet, or can be set directly in code. Four other properties that appear on the first page of the control's property sheet are worthy of specific mention:

- *Shareable*: Determines whether more than one program can access the same multimedia device. Set to *False*. (Note: We confess that we are unable to imagine a scenario where you would ever need it to be set any other way. The types of devices supported by this control do not lend themselves to being controlled by more than one user (or program) at a time anyway.)
- *Silent*: Determines whether sound is played or not. Set to *False*. Allows you to implement a "*Mute*" function for external devices simply by setting it to *True* at run time. (Note: It does **not** mute files that are being played through either the *WaveAudio* or *Sequencer* devices. It only suppresses the audio component from other devices like a CD or Digital Video.)
- *Enabled*: This control-level property overrides the settings of the individual buttons. When set to False, all visible portions of the control are disabled irrespective of the settings of the individual buttons. Set to *True*.
- *AutoEnable*: Determines whether the control automatically enables those buttons that are applicable to the current mode of the selected DeviceType. Set to *True*.

Obviously, one way of using this control is to place it on a form with whatever buttons are needed made visible and allow the user direct control over the functionality. (In fact, it is perfectly possible to write your own personalized "Media Player" in Visual FoxPro using this control, though it hardly seems worth the effort.) However, in the context of an application, the most likely use for the control would be as an invisible object whose functionality is controlled from within the code. For this reason our subclass (*CH09.vcx::xMMedia*) of the MultiMedia control has all of the buttons enabled, but they have been set up as invisible.

Using the MultiMedia control (*Example: frmMMedia.scx*)

The sample form (**Figure 23**) illustrates how the MultiMedia control might be used in a form to play back the selected item.

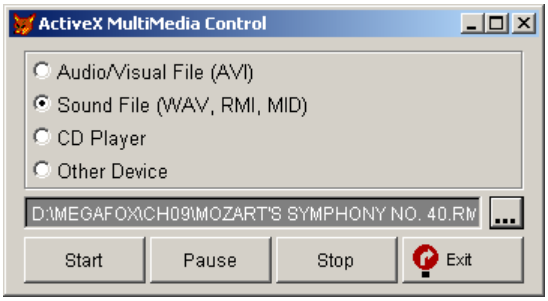


***Figure 23**. Using the MultiMedia control (frmMMedia.scx).*

In order to initiate playback, the *DeviceType* and *FileName* properties of the MultiMedia control must be set correctly. The form handles the first part of this using two custom properties (*cDevType* and *cFileType*), which are set explicitly by the *InterActiveChange()* method of the option group control.

Selection of the file name is handled by the form's custom *GetPlayFile()* method, which is called by clicking the "expansion" button to the right of the file name textbox. This method uses the setting of the form's *cFileType* property to determine whether a **GETFILE()** dialog should be displayed (playing AVI or music files), or a **GETDIR()** dialog (for the CD Player), or whether the textbox should be enabled for direct entry (Other Device).

The actual playback is handled by the form's custom *PlayStart()* method. This starts by checking that a file name and device type have been specified and re-initializing the Pause button's caption.

```
LOCAL lcDevice
WITH ThisForm
  *** Do we have a filename?
  lcDevice = .txtPlayFile.Value
  IF EMPTY( lcDevice )
    MESSAGEBOX( "No Device has been specified", 16, "Nothing to do" )
    RETURN
  ENDIF
  *** Get the device type
  lcDevType = .cDevType

  *** First, reset the Pause button caption
  .cmdPause.Caption = "Pause"
```

Next, we set the properties on the local instance of the MultiMedia control. Note that we need to check the device type in order to set it correctly, depending on the type of file selected. Attempting to playback an mpeg file using the "*WaveAudio*" device will not work.

```
  WITH .oMPlayer
    *** Set Filename
    .FileName = lcDevice
    *** Set Device type correctly
    IF lcDevType = "Waveaudio"
      *** We need to set the device type correctly
      *** for the type of sound file chosen
      lcDevType = IIF( JUSTEXT( lcDevice ) = "WAV", "Waveaudio", "Sequencer" )
    ENDIF
    .DeviceType = lcDevType
```

Depending on the status of the device, we must either open it (if it's not already open) or reset it to the beginning (if the device is already open). This is handled by calling the control's *Command()* method and passing the appropriate instruction. Finally, the *Command()* method is called with the "Play" instruction to start playback.

```
    *** Now we need to check the status of the device
    IF ThisForm.lDevOpen
      *** Already open, just "re-wind"
      .Command = "Prev"
    ELSE
      *** Open it and set the flag
      .Command = "Open"
      ThisForm.lDevOpen = .T.
    ENDIF
    *** Ensure that playback is asynchronous
```

```
    .Wait = .F.
    *** Now play!
    .Command = "Play"
  ENDWITH
ENDWITH
```

The *Command()* method can accept a range of instructions, which are detailed in **Table 10**. Each instruction uses one or more properties on the control, which need to be set before calling the method. The table also shows the default values that are used when nothing is specified.

***Table 10**. Multimedia Command() method options.*

Command	Description	Uses (*Default*)
Open	Opens the entity specified in the *FileName* property for playback.	Notify (*False*) Wait (*True*) Shareable (*False*) DeviceType FileName
Play	Initiates playback on the currently open device.	Notify (*False*) Wait (*True*) From To
Pause	If supported, pauses playback (or recording). If executed while the device is paused, tries to resume the operation.	Notify (*False*) Wait (*True*)
Stop	Stops the current operation on the current device.	Notify (*False*) Wait (*True*)
Back	If supported, steps backward on the current device.	Notify (*False*) Wait (*True*) Frames
Step	If supported, steps forward on the current device.	Notify (*False*) Wait (*True*) Frames
Prev	Goes to the beginning of the current playback track. If executed within three seconds of another "*Prev*" command, goes to the beginning of the previous track (if at first track, or there is only one track, always goes to the beginning).	Notify (*False*) Wait (*True*)
Next	Goes to the beginning of the next track (if at last track, goes to beginning of last track).	Notify (*False*) Wait (*True*)
Seek	If not playing, goes to the location specified in the To property. If playing, jumps to and resumes playing from the specified position.	Notify (*False*) Wait (*True*) To
Record	Initiates recording on devices that support the operation.	Notify (*False*) Wait (*True*) From To RecordMode (*0-Insert*)
Eject	Ejects media on devices that support the operation.	Notify (*False*) Wait (*True*)
Save	Saves an open file on devices that support the operation.	Notify (*False*) Wait (*True*) FileName

Full details of these properties and their values can be found in the Help file for the control (MMEDIA98.CHM). The only one that we need to concern ourselves with immediately is the *Wait* property. This determines whether the control executes the next command synchronously or asynchronously. However, it only affects the *next* command to be executed, which is why we have to explicitly set it each time we initiate the playback.

The other methods are all extremely simple; the only thing that we must point out is that it is imperative that a device be explicitly "closed" before attempting to reset the control's *DeviceType* property. This is why the custom *CloseDevice()* method was added:

```
WITH ThisForm
  WITH .oMPlayer
    .Command = "Stop"
    .Command = "Close"
  ENDWITH
  *** Set the device flag to closed
  .lDevOpen = .F.
ENDWITH
```

We call this method variously from the *GetPlayFile()*, *PlayStop()*, *SetDevice()* and *Destroy()* methods. We found that failing to do so would cause VFP to crash with remarkable regularity—in this case it is definitely better to be safe than sorry.

*One other Command() method instruction is listed in the Help for the MultiMedia control. The "*Sound*" instruction purportedly "plays a sound by using the MCI_SOUND command". However, the MCI documentation has no reference for MCI_SOUND and, in the control, the instruction does not appear to do anything at all. As far as we can tell, this is a documentation error in the Help file, and may be a reference to a command that has been removed in later versions of the API.*

How do I use other types of media in my application?

As we have seen, both the Animation and the MultiMedia controls are limited in the types of media that they can handle. Admittedly the latter is really intended for accessing physical devices rather than simply playing video or music files, so maybe we should not complain too much. However, when it comes to dealing with some of the newer forms of media (MP3 or MPEG) files, neither of these controls will do. Apart from any proprietary controls (which are outside the scope of this book), there is little we can suggest other than to use the Windows Media Player (contained in MSDXM.OCX, associated Help file WMPLAY.CHM).

There are several things to note about this control, before we get down to trying to work with it. First, the `ocx` actually contains *two* versions of the control:

- Version 6.4, which is intended for use in desktop applications
- Version 7.1, which is intended for use in Web pages

The Help file also contains documentation for both versions; however, in this chapter we are only dealing with Version 6.4.

Second, this control does not ship with Visual FoxPro. It is (optionally) installed as part of the Windows installation process and so there is a significant risk that it may not be available on an end user's machine or that, even if it is available, it may be a different version from the one you have used.

Third, unlike most other ActiveX controls, the Media Player does not expose a properties sheet of its own. The only access to its properties is through the All tab of the Visual FoxPro properties sheet.

Finally, this control is far more complex than any of the ActiveX controls that we have discussed previously. It has a great deal of built-in functionality and exposes many properties and methods.

Subclassing the Media Player control (*Example: CH09.vcx:: xMPlayer*)

Unfortunately, this control does not seem to work well in the Visual FoxPro environment and, although we could instantiate the control as a non-visual object, we could not manage to get it work at all. Every attempt to do so resulted in an "*Unknown COM status code*" error. The following code, run from the command line, should initiate playback of the specified file but doesn't:

```
ot = CREATEOBJECT( 'mediaplayer.mediaplayer.1' )
ot.AutoStart = .T.
ot.Filename = 'STARSFULL.MP3'
ot.Play()
```

Using the Media Player control (*Example: frmMPlayer.scx*)

In fact, the only way in which we could get the control to function at all was as a visual object included in a form (see **Figure 24**). The sample form uses a subclass of the Media Player ActiveX control to play back a file. The normally displayed controls for the Media Player have been hidden by setting the control's *ShowControls* property to *False*, and the use of the mouse to stop and start playback by clicking on the control has also been disabled by setting the *ClickToPlay* property to *False*.

Note that the control determines how it should be displayed depending upon the type of file that has been selected for playback. The appearance in Figure 24 is with an MPEG file selected, but had we selected an MP3 file instead, the display area would be hidden and all you would see would be a blank space on the form. This could, of course, be handled by adding code to resize the form, or to bring a static image to the front when a non-visual file is selected, but these refinements are being left for you, the reader, to implement according to your own requirements.

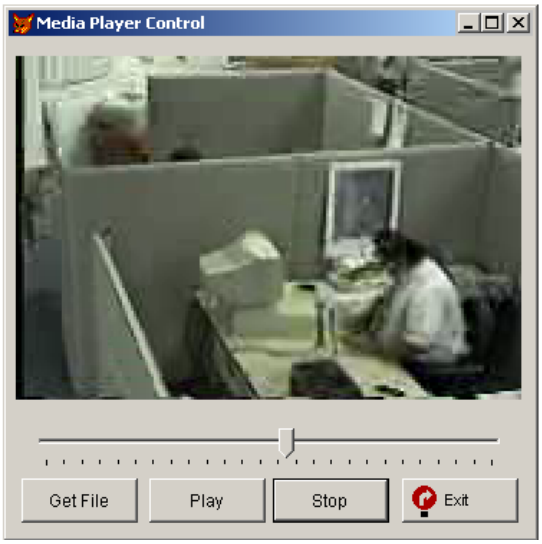


***Figure 24**. Using the Media Player (frmMPlayer.scx).*

Inside the form, the Media Player object is controlled by calling its *Play()* and *Stop()* methods as necessary from the form's custom *PlayStart()* and *PlayStop()* methods. These two methods use the value returned by the *ReadyState* property to determine whether to implement the requested action as follows:

```
*** PlayStart Method Code:
WITH ThisForm.oMPlayer
  IF .ReadyState # 4
    *** Not ready to start playback
    MESSAGEBOX( "Unable to start playback", 16, "Not Initialized" )
    RETURN
  ENDIF
  .Play()
ENDWITH

*** PlayStop Method Code:
WITH ThisForm.oMPlayer
  IF .PlayState # 2
    *** Not playing anyway - do nothing
    RETURN
  ENDIF
  .Stop()
ENDWITH
```

The Media Player control makes extensive use of this, and other, "state" properties, and the constants returned from these properties, together with their meanings, are listed in the MPLAYCONST.H file included with the sample code for this chapter.

Note that the playback volume is controlled in this example using another ActiveX control, the Slider, whose *Scroll()* event is used to change the Media Player's *Volume* property

directly. Once the Slider has focus it can be controlled by dragging with the mouse, clicking at any point on the scale, or by using the keyboard left and right cursor keys.

This very simple example shows, yet again, just how little code is needed to tap into the functionality provided by ActiveX controls and, while this particular control may not be as immediately useable in an application environment as some others, in the right situation it may still provide a good solution.

How do I add a status bar to a form? (*Example: frmSbastd.scx*)

The status bar ActiveX control is another of the controls contained in the common controls library MSCOMCTL.OCX and is described in detail in the associated Help file CMCTL198.CHM. The control can be used to give a form the standard "Windows style" paneled status bar (see **Figure 25** for an example).

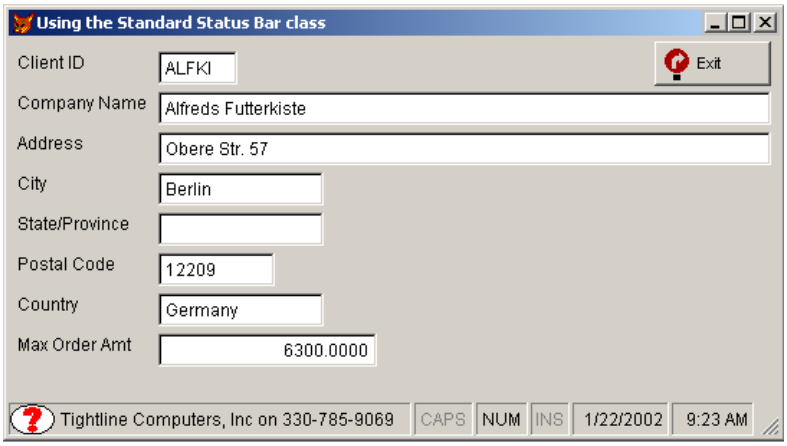


***Figure 25**. The standard status bar control in a form (frmsbastd.scx).*

As with all the other ActiveX controls that we have discussed in this chapter, the first thing we did was to create a subclass of the control (see the *xsba* class in CH09.VCX). The location of the status bar on the form is controlled by its *Align* property and it can be displayed at the top, left, right, or, more conventionally, across the bottom of a form. The default value for this property is *0 (None)* so in our subclass we explicitly set it to *2 (Bottom)*. The control is a container for "*panel*" objects that can be displayed in one of two ways, controlled by a *Style* property.

- *Style = 1 (Simple)* defines a single, full-width panel that can display only the text specified by its *SimpleText* property. Nothing else can be displayed in this style.
- *Style = 0 (Multiple Panels)* activates the Panels collection, which allows for up to 16 separate panels to be defined, and whose allowable contents are controlled by their individual *Style* properties.

The status *Panels* collection conforms to the standard ActiveX model and has a *Count* property to track the number of panels. You can use either the *Index* or the *Key* property to

access contained panel objects. The *PanelClick()* event receives, as a parameter, an object reference to the panel over which a mouse click was detected and so can be used to change either the appearance or contents of the control at run time if required. The panel objects have a set of properties as described in **Table 11**.

***Table 11**. Status bar panel properties.*

Property	Type	Comments
Alignment	Numeric	0 = Left, 1 = Center, 2 = Right
AutoSize	Numeric	0 =None means that the panel does not get resized at runtime. 1= Spring means that any extra status bar space is divided equally among all panels with this setting when the status bar is instantiated. At least one panel should always be defined with this setting to ensure proper display. 2 = Contents means that the panel gets resized at runtime depending on what it is displaying at the time. This causes the panel size to vary as its content changes and the visual effects can be distracting.
Bevel	Numeric	0 =None, 1= Inset, 2 = Raised
Enabled	Logical	
Index	Numeric	Panels Collection index number
Key	Character	Panels Collection key string
Left	Numeric	Left position inside the Status Bar in Pixels
MinWidth	Numeric	Minimum Width for the panel in Pixels (Default = 10)
Picture	Character	Name of the picture to display in the panel (no control over location)
Style	Numeric	Panel Contents 0 = Text (read from Text Property) 1 = Caps Key Setting 2 = Num Lock Key Setting 3 = Insert Key Setting 4 = Scroll Lock Key Setting 5 = System Clock 6 = System Date 7 = Kana Lock Setting (Japanese Operating Systems only)
Tag	Character	Additional data store—not used natively by the control
Text	Character	Text to display
ToolTipText	Character	ToolTip text to display
Visible	Logical	
Width	Numeric	Actual panel width, in pixels

There is one, rather peculiar, bug with this control. If your form is defined as a "Top Level" Form (that is, ShowWindow =2), the drag bars for sizing the form that are normally shown at right of the status bar simply disappear when the form is run.

Setting up a standard status bar (*Example: CH09.vcx::sbastd*)

If you need to give your forms a consistent look and feel, you will want to create a standard status bar class and this is perfectly simple to do. The "standard" status bar class defines six panels as illustrated in Figure 25. When not in simple mode, the properties sheet for the

control allows you define the settings for individual panels on the second page of the pageframe—otherwise, any settings on this page are simply ignored.

Notice that in this class the first panel has been set up to use the "*spring*" setting for sizing (*AutoSize = 1*). This ensures that the other panels, which all use fixed width (*Autosize = 0*), are displayed correctly and that the only panel to resize when the form is resized is the first (unless the form width is reduced so that there is insufficient space for the fixed-width panels to display correctly).

The example form requires no code at all, because the text in the first panel is static and was defined (along with the picture) directly in the class using the control's own Property sheet. However, in order to accommodate dynamically changing text, the *xsba* class (on which the standard toolbar class is based) defines a custom method named *SetText()* as follows:

```
LPARAMETERS tcText, tnPanel
LOCAL lcText, lnPanel
*** If no text passed, assume an empty string to clear the panel
lcText = IIF( VARTYPE( tcText ) = "C" AND NOT EMPTY( tcText ), tcText, "" )
IF This.Style = 1
  *** We are in Simple Text mode
  *** just set whatever was passed
  This.SimpleText = lcText
ELSE
  *** We are in multiple panel mode. Default to first panel if not passed
  lnPanel = IIF( VARTYPE(tnPanel) = "C" AND NOT EMPTY(tnPanel), tnPanel, 1 )
  This.Panels[lnPanel].Text = lcText
ENDIF
```

This method accepts a text string and a panel index that only needs to be specified if not setting Panel 1. It then sets the appropriate source property (either *Text* or *SimpleText*) as defined by the style.

What's the point of the simple style status bar? (*Example: frmSbabase*)

The simple style is, admittedly, very limited indeed. As noted earlier, it can have only one panel and that panel can only display plain text. However, that is precisely the functionality of the VFP status bar that is used to display the contents of a VFP control's *StatusbarText* property. The biggest problem with using this is not that it is in any way difficult, but in training end users to look at the bottom of the screen for messages and prompts rather than on the current form. Using a status bar with the Simple Style is an ideal way to address this and provide "in-form" prompts and messages that are linked to whichever control has the focus (see **Figure 26**).

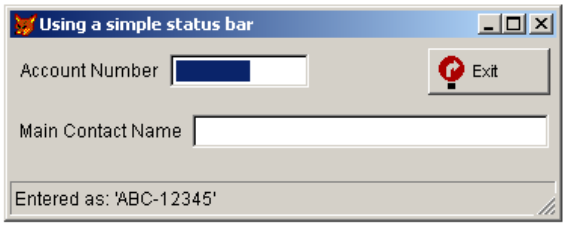


***Figure 26**. A simple status bar for displaying dynamic text (frmsbabase.scx).*

In the example form code, a subclass (*xsbasimple*) of the status bar root class is used to define a status bar using the simple style. Code has been added to the *GotFocus()* and *LostFocus()* events of the textboxes to ensure that the contents of each control's *Comment* property is posted to the status bar as focus moves. Of course, in practice we would create a special set of subclasses to work with a status bar in this way rather than relying on instance-level code.

Note that we have chosen to use the comment property as the source for the text, rather than create a custom property. This is so that we can define the text for bound fields directly in the database container. (Remember that on the Field Mapping tab of the Options dialog you can specify whether fields created in forms by dragging from the Dataenvironment should include *Caption*, *InputMask*, *Format*, and *Comment* values from the DBC.) Since we all, always, fill in the comments for fields in the DBC (*we do all do this*, *don't we?*), it provides an easy way to pass on those comments directly to our users.

The code required is very simple thanks to the custom *SetText()* method that we defined in the *xsba* root class. To the *GotFocus()* event we have added:

```
DODEFAULT()
ThisForm.oSba.SetText( This.Comment )
```

which sets the status bar text on entry, and then in the *LostFocus()* we clear any text by calling the *SetText()* method with no parameters at all:

```
DODEFAULT()
ThisForm.oSba.SetText()
```

You could, of course, achieve the same result by creating a textbox class and sizing it to the form; however, the status bar does offer one additional feature. Even the simple style is self-sizing to whatever form it is added to, and it does provide a Windows style "drag" region at the bottom right corner of the form, which VFP forms do not otherwise include.

Managing the status bar dynamically (*Example: frmSbacus*)

You may prefer, rather than defining and using a standard status bar for all forms, or just using a simple status bar, to allow your users to define how the status bar in their individual forms should appear. To do this you would need to create a storage mechanism to hold individual users' preferences (a simple local table would do), and some loader code in your form class to ensure that the status bar is set up accordingly. The basic principles are illustrated by the example form (see **Figure 27**), which allows you to dynamically configure the status bar on the form.

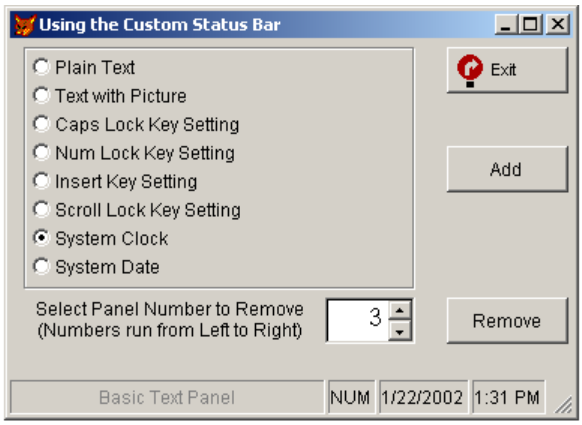


***Figure 27**. Changing the status bar at run time (frmsbacus.scx).*

This form uses another subclass (*xsbacustom*) of the *xsba* root class. This subclass includes two additional methods for adding and removing panels from the status bar at run time. The *AddPanel()* method expects to receive a parameter object that defines the required parameters for adding a new panel, and the first part of the code ensures that the parameter is at least of the correct class.

```
LPARAMETERS toPanel
*** Must pass a panel setting object
IF VARTYPE( toPanel ) # "O" OR LOWER(toPanel.Class) # 'xpanel'
  *** Bale out right here!
  RETURN .F.
ENDIF
WITH This
  *** Are there any valid (enabled) panels?
  IF This.Panels.Count = 0
    lnPanelId = 1
  ELSE
    *** Assume the next highest number
    lnPanelID = This.Panels.Count + 1
  ENDIF
```

Next, the number of existing panels is checked, and the index for the new panel to be added is determined. It is then simply a matter of calling the control's *Panels* collection's *Add()* method and passing the required index number. Note that panels are added from left to right, using their index numbers, and that we use the index number to define the Key for the panel. (Doing it this way makes it much easier to write generic code to remove panels.) It is then simply a matter of setting the rest of the properties:

```
*** And add the panel
  This.Panels.Add( lnPanelid )
  loNewPanel = This.Panels[ lnPanelID ]
  WITH loNewPanel
    *** Set the Key using PanelID to ensure Uniqueness
    .Key = "Panel" + TRANSFORM( lnPanelID )
    *** Get Other Settings from the object
```

```
    .Alignment = toPanel.xAlignment
    .AutoSize = toPanel.xAutoSize
    .Bevel = toPanel.xBevel
    .Enabled = toPanel.xEnabled
    .MinWidth = toPanel.xMinWidth
    .Picture = toPanel.xPicture
    .Style = toPanel.xStyle
    .Tag = toPanel.xTag
    .Text = toPanel.xText
    .ToolTipText = IIF( EMPTY( toPanel.xToolTipText ), ;
                        toPanel.xText, toPanel.xToolTipText )
    .Visible = toPanel.xVisible
    .Width = toPanel.xWidth
  ENDWITH
  .Visible = .T.
ENDWITH
```

The parameter object is defined, using a line base class, as the *xpanel* class in CH09.VCX. This class simply defines properties for each parameter of the panel object. These get populated in the calling method, which in this example is the *OnClick()* method of the form's Add button.

Removing a panel is also quite straightforward. Simply use the spinner to define which panel you wish to remove and click the form's Remove button. The required panel index number is passed on to the class *RemPanel()* method as a character string, which is used to generate the key name for the panel in question and return its internal index. The *Panels* collection's *Remove()* method is then called, passing the correct index number:

```
LPARAMETERS tcKey
LOCAL lnIdx, lnCnt
IF VARTYPE( tcKey ) = "C" AND NOT EMPTY( tcKey )
  lnIdx = 0
  FOR lnCnt = 1 TO This.Panels.Count
    IF This.Panels[lnCnt].Key == tcKey
      lnIdx = This.Panels[lnCnt].Index
      EXIT
    ENDIF
  NEXT
  IF NOT EMPTY( lnIdx )
    This.Panels.Remove( lnIdx )
  ENDIF
ENDIF
```

Next we have to reset the Key values of any remaining panels so that they are synchronized with the index number. Of course, if there are no panels left, we can simply make the status bar invisible.

```
*** If last panel goes, hide the bar
IF This.Panels.Count = 0
  This.Visible = .F.
ELSE
  *** Otherwise re-synch the keys and index
```

```
  FOR lnCnt = 1 TO This.Panels.Count
    This.Panels[ lnCnt ].Key = "Panel" + TRANSFORM( lnCnt )
  NEXT
ENDIF
```

Conclusions about the status bar control

With the single exception of losing the sizing handle when the form on which it resides is defined as Top Level form, the status bar is a well behaved and flexible tool. This section has shown several different ways in which it can be used to enhance the appearance of your applications and make them look even more "professional".

What is the Winsock control?

The Winsock control (contained in MSWINSCK.OCX, associated Help file MSWNSK98.CHM) is a "*socket*" object that allows you to connect to a remote machine and exchange data using either of two communication protocols:

- User Datagram Protocol (UDP): A connectionless protocol (analogous to passing a note) under which data is passed from one "peer" computer to another without the need to establish an explicit connection.
- Transmission Control Protocol (TCP): A connection-based protocol (analogous to a telephone call) that requires that a connection be explicitly established between one computer acting as the "client" and another computer acting as the "host".

So which protocol is best?

The answer, as so often in VFP, is that *it depends*. Each protocol has its benefits, and its drawbacks. **Table 12** gives a side-by-side comparison.

***Table 12**. Features of the protocols available to the Winsock control.*

UDP	TCP
No connection required. Participating computers are equivalent to each other and each binds to the remote port of its peer.	Explicit connection required. One participant (the "host") must have an active listener so that a client can establish a connection.
Data must be transmitted in a single send operation. Maximum size is determined by the network setup and excess data is simply lost.	Data can be transmitted using multiple send operations. Maximum size is not dependent upon network configuration.
A single socket can switch between partners without needing to reset.	A single socket can only handle one connection at a time. Therefore to change partners, any existing connection must be closed before another can be established.
No acknowledgements.	Explicit connection allows query/acknowledgement to control communication.
No connection and no integrity checking.	Connection is managed and data integrity ensured.
Lower resource requirement, unreliable.	Higher resource requirement, reliable.

It is apparent from Table 12 that the UDP protocol is better suited to applications that either involve manual intervention (for example, Instant Messaging) or are simply broadcast without requiring any acknowledgement. Conversely, TCP is a better choice when an automated process is required (for instance, Error Reporting) or when any form of exchange is required.

How do I include messaging in my application? (*Example: IMDemo.prg*)

The easiest way to build messaging into an application is to use the Winsock control and UDP protocol. As noted earlier, UDP does not require a connection to be established between participating machines. Instead, each machine must create a socket and set five properties, as follows:

- *Protocol* Set to 1 for UDP protocol.
- *LocalHostName* Sets itself automatically to the local machine name when the control is instantiated (note that the *LocalIP* property is also set to the host machine's IP Address automatically).
- *LocalPort* This is the port on which the socket will receive incoming messages. Defaults to 0. Remote machines must set their *RemotePort* properties to point to this port.
- *RemoteHost* This is set to point to the machine name for remote participant. The example code sets this to the same as the *LocalHostName* unless another name is specified. (Note that the remote machine's IP Address can, alternatively, be set in the *RemoteHostIP* property.)
- *RemotePort* This is the port on the remote machine to which data will be sent. Obviously it must correspond to whatever is defined as the "local" port on that machine.

To activate the socket, all that is required is to call the *Bind()* method, passing the local port that has been defined. This reserves the specified port for UDP communications and prevents other applications from accessing it. It is important, therefore, when using UDP to ensure that you do not choose a port that is already defined for other purposes. (Note: A list of port assignments for commonly used services can be found in the Windows Resource Kit or on the MSDN Web site by searching for "TCP and UDP Port Assignments".)

Once the port has been bound, data can be sent (to whatever machine is identified by the *Remote* properties) by simply calling the *SendData()* method and passing the message as a text string. When an incoming message is received, the *DataArrival()* event fires and code in the associated method can be used to deal with the message. The example uses two forms that simulate a simple messaging service (see **Figure 28**) using edit boxes to enter and display message text.

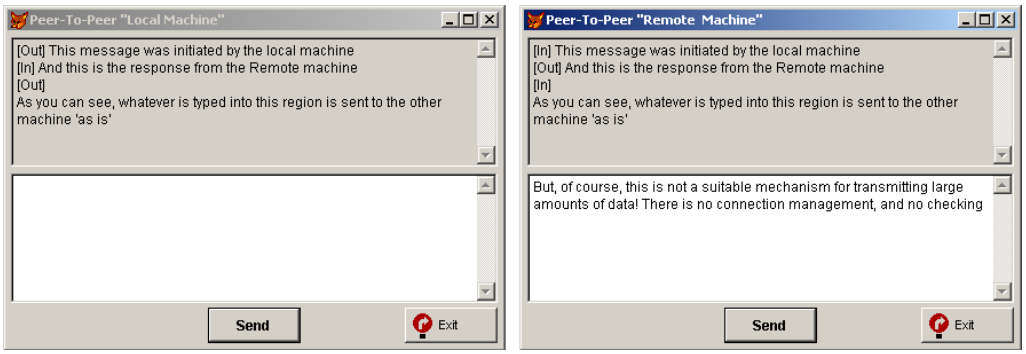


***Figure 28**. Simple messaging demonstration (frmplocal.scx and frmpremote.scx).*

Setting up a messaging form

The two forms are, essentially, mirror images of each other. The "*remote*" property settings for the "Local Machine" are identical to the "*local*" settings for the "Remote Machine" and vice versa. The code in both forms is almost identical too. The only significant difference between them is the names of the objects they contain. Each form has an instance of the Winsock control that has been set up to use the UDP protocol (*Protocol = 1*) and whose *DataArrival* event has been modified to call the form's custom *ReadData()* method as follows:

```
LPARAMETERS bytestotal
ThisForm.ReadData()
```

The code shown next is from the "local" form (FRMPLOCAL.SCX). The form's *Init()* method is used to set up the socket. Up to three parameters may be passed (the remote server name, remote port number, and local port number), although we have also defined default values for all three. The connection is then initialized using these values.

```
LPARAMETERS tcServer, tnRemPort, tnLocPort
LOCAL lcServer, lnRemPort, lnLocPort

*** Default values if nothing specified
*** Server Name (Local by default)
lcServer = IIF( EMPTY( tcServer ), ;
               LEFT( SYS(0), AT( "#", SYS(0)  ) - 1 ), tcServer )
*** Remote Port ID
lnRemPort = IIF( EMPTY( tnRemPort ), 1001, tnRemPort )
*** Listening Port ID
lnLocPort = IIF( EMPTY( tnLocPort ), 1002, tnLocPort )
```

```
*** Set up the winsock connection for UDP
WITH ThisForm.oPLocal
  .Protocol = 1
  .RemoteHost = lcServer
  .RemotePort = lnRemPort
  .Bind( lnLocPort )
ENDWITH
```

The form itself is trivial. We have two edit boxes, one for entering text to be sent, and one (read-only) for displaying text. The Send button calls the form's custom *Send()* method:

```
LOCAL lcText
WITH ThisForm
  *** Send the text
  lcText = ALLTRIM( .edtOutward.Value )
  IF NOT EMPTY( lcText )
    *** Transmit the data
    .oPLocal.SendData( lcText )
    *** Add the text to the display
    .ShowData("[Out] " + lcText )
    *** Clear the send box
    .edtOutward.Value = ""
  ENDIF
ENDWITH
```

This retrieves whatever has been entered into the outbound edit box and passes it on to the socket's *SendData()* method. An "*Out*" prefix is then added to the message and the result posted to the display by calling the form's custom *ShowData()* method.

Incoming messages are dealt with in the custom *ReadData()* method, which is called from the *DataArrival()* event of the socket. This initializes a string buffer and calls the socket's native *GetData()* method to read the message into the specified buffer, which must be passed by reference. Note that the *DataArrival()* event does receive the number of bytes in the incoming message as a parameter. That value could be passed on and used to initialize the buffer correctly but, since we are using a local variable here, there is no *requirement* to do it.

```
LOCAL lcText
WITH ThisForm
  lcText = ""
  .oPLocal.GetData( @lcText )
  *** Add the Prefix
  lcText = "[In] " + lcText
  .ShowData( lcText )
ENDWITH
```

Having received the incoming text, and added an "*In*" prefix to it, the form's custom *ShowData()* method is called to display the text. This method sounds the system bell when a message is about to be posted and adds the message to the display.

```
LPARAMETERS tcText
LOCAL lcData, lnNewLine
IF ! EMPTY( tcText )
  ?? CHR(7)
  WITH .edtInWard
```

```
    lcData = ALLTRIM( .Value )
    lnNewLine = LEN( lcData ) + 2
    lcData = lcData + IIF( NOT EMPTY( lcData ), CHR(13), "" ) + tcText
    .Value = lcData
    .SelStart = lnNewLine
    .SetFocus()
  ENDWITH
  .edtOutWard.SetFocus()
ENDIF
```

Extending the example

The example here is deliberately simplistic because we are really focusing on how to use the control rather than discussing the design of a full-blown messaging application. However, one of the benefits of using the UDP protocol is that it is possible to switch remote machines at any time by simply resetting the *RemoteHost* property on the control. So, to implement a messaging application, all that is needed is a list of users and their associated machine IDs, and an interface that allows you choose a person to communicate with.

Another possible use for the UDP protocol would be to enable your system administrator to send messages to users who are currently logged into an application. In that scenario you would simply instantiate a socket at application startup and set it to communicate with a central server. To broadcast a message, all that is needed is to loop through a list of users, reset the *RemoteHost* property, and send the same message to each. On the end user's side the *DataArrival()* event is used to display the message.

Since UDP does not require a connection, only those users currently logged in would receive the message. However, attempting to send a message to a UDP host that is not available will still generate an OLE exception error (usually VFP error number 1429). The example form's error event includes code to trap OLE errors and display the contents of the `AERROR()` array in a message box. However, in a broadcast system, the error could be used to verify the list of users who are logged. (No error means that the user really *is* logged in.)

How do I transmit error reports without using e-mail?

As noted in Chapter 4, one way of collecting automated error reports is to use e-mail, but, for an in-house application running over a local network, we can also use the Winsock control and the TCP protocol. Using TCP is rather more complex than using UDP because we need two different types of object, a client and a server. First, we need a client whose function is to initiate communications by requesting a connection from the server. It then has to wait for a response, before transmitting its data. The client in our example (TCPCLIENT.PRG) exposes a *PostData()* method, which attempts to establish a connection and, if successful, transmits whatever data has been passed to it.

Second, we need a server whose function is to monitor a specific port for connection requests and act upon them. Since the objective of this particular server is to receive error reports transmitted from other machines, it must be able to accept multiple simultaneous connections. However, a single socket can only handle one connection at a time, so in order to handle these multiple requests, we must instantiate one socket as a "*listener*". The listener's function, whenever a client request is detected, is to instantiate a new socket and pass it the new connection ID so that it can handle things from that point on (see **Figure 29**).

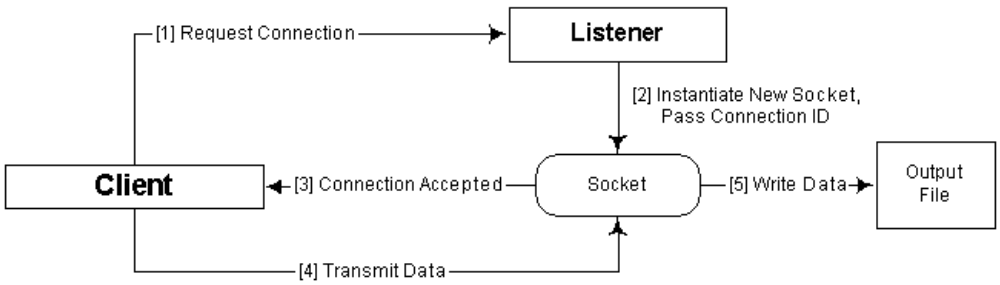


Figure 29. *Using the Winsock control for messaging.*

Thus we need two different configurations for the Winsock control. First we need the listener, which is defined in the subclass *xWSListener*, and then we need one to accept a connection and do whatever is necessary. In this example we want whatever data is transmitted written out to a file, and so we have created a subclass named *xWSLogFile*. Both of these subclasses inherit from the *cntWinsock* class, which adds the basic OLE Container subclass (*xWinSock*) to our standard container root class.

The reason for this (rather convoluted) structure is that one of the limitations of Visual FoxPro is that the only classes that can instantiate an OLE Container control at run time are Forms and Toolbars, and they can only do it by using *AddObject()*. However, although these classes do have a *RemoveObject()* method, it does not remove the property that is created when an object is added; it merely sets it to a *Null* value. This makes it much harder to manage the sockets collection without creating a vast number of "*used-once*" properties on our server object. By adding the *OLEContainer* to a standard VFP container at design time (which we *can* do), we can then instantiate it at run time using `CREATEOBJECT()`. This makes it much simpler to manage the server's sockets collection.

> ! *WARNING! There is a problem when attempting to run the TCP example on a single machine that appears to be due to a timing conflict within VFP. The example code works perfectly well when the client and the server objects are on physically separate machines, but, when both are on the same machine, the client is unable to establish a connection to the server. However, if a SET STEP ON is placed in the client code immediately before the attempt to connect, and the debugger is used to step through the actual connection, everything succeeds! This is what makes us think this is a timing issue. We were unable to find a workaround that would enable the client to connect when the server was on the same physical machine without that explicit SET STEP ON.*
>
> *We would like to thank Andy Goeddeke and Viv Phillips for their help in investigating and confirming this behavior under a variety of operating systems including NT4, W2K Professional, W2K Server, and Windows XP Professional.*

The client definition (*Example: TCPClient.prg*)

Our example client is defined programmatically (it has no visual component after all) and is based on the native Custom base class. Three custom properties are used, one for the instance

of the socket and one each for the name of, and port to connect to, the remote server. By default, the remote server is set to point to the local machine, and the port is pre-defined as 1001. However, the *Init()* method can accept, as parameters, values that will override the server and port defaults:

```
FUNCTION Init( tcServer, tnRemPort )
LOCAL lcServer, lnRemPort
  WITH This
    *** Default to local machine name if nothing specified
    .cServer = IIF( EMPTY( tcServer ), .cServer, tcServer )
    *** Remote Port ID defaults to 1001
     .nRemPort = IIF( EMPTY( tnRemPort ), .nRemPort, tnRemPort )
     *** Create the Connection object
     .oConnect = CREATEOBJECT( "MSWinsock.WinSock" )
  ENDWITH
ENDFUNC
```

Other than the *PostData()* method, which we will discuss in detail in a moment, the client only defines two more methods. The first checks for, and closes, the connection if it is open (named, unsurprisingly, *CloseConnection()*). The second modifies the native *Destroy()* method to call *CloseConnection()* so that we do not inadvertently leave connections dangling. All of the real work is done in the exposed *PostData()* method.

The first thing is to ensure that the current connection is closed because, unlike under UDP, when using TCP we cannot simply reset the remote host parameters while the connection is open. Then we reset the connection parameters for the server that we want to connect to:

```
  FUNCTION PostData( tcData )
    LOCAL lnCnt, lnState, llSend, llTxOK
    *** Must ensure that the connection is closed before setting host
    This.CloseConnection()
    WITH This.oConnect
      *** Set server parameters
      .RemoteHost = This.cServer
      .RemotePort = This.nRemport
```

To initiate the connection process, we simply call the socket control's native *Connect()* method. This uses the current settings for the remote host and requests a connection. (*If you are running this example on a single machine, you will need a* ***SET STEP ON*** *immediately before this method call.*)

```
      *** And initiate the connection
      .Connect()
```

In order to determine whether the connection request has been accepted, we need to check the *State* property of the socket. Since there may be a delay in confirming the connection, this needs to be done inside a loop as follows:

```
      *** Poll status for connection
      lnCnt = 1
      DO WHILE .T.
```

```
      lnState = .State
      *** Allow some time to connect before checking
      INKEY(0.3,'hm')
      *** If we are connected, get out now
      IF lnState = 7
        llSend = .T.
        EXIT
      ENDIF
      lnCnt = lnCnt + 1
      *** Break out if we get to 100, otherwise we're stuck forever
      IF lnCnt > 100
        EXIT
      ENDIF
      DOEVENTS()
    ENDDO
```

The various values returned by the *State* property can be found in the WSOCKCONST.H file that is included with the sample code for this chapter. The one we are interested in here is *7*, which tells us that the connection has been established and that the server is ready for us to send data. So, as soon as we get a confirmed connection, we set the *llSend* flag and exit from the loop. (Note the "break out condition"—without that we could find ourselves in an endless loop here!)

Assuming we got a connection, we can then transmit whatever data was passed to the client and set the result flag, which will be returned from this method.

```
    *** If we got a connection, send the data
    IF llSend
      .SendData( tcData )
      llTxOK = .T.
    ENDIF
  ENDWITH
  RETURN llTxOK
ENDFUNC
```

This is an extremely simple client, and much more could be done if, rather than simply sending data, we wanted to exchange data with the server. However, for the purposes of sending an error report this is all that is needed. The server code already illustrates how to handle receiving data and, where data exchange is required, the client class would need similar methods and appropriate code.

The server definition (*Example: CH09::xtcpServer*)

As noted earlier, the TCP server is constructed rather differently from the client and uses two different subclasses of the Winsock control. The server itself is a form class whose *Visible* property has been hidden and set to *False*, and whose *Show()* method has been disabled, to prevent it from being made visible. Each of the socket subclasses is based on our containerized subclass of the Winsock control, *cntWinsock*. This has its *Protocol* property set to 0 -TCP Protocol and the *LocalPort* defined as 100.

In addition, five of the socket's native methods have been surfaced in the container by adding custom methods of the same name. This avoids having to call the socket's methods directly from external objects and allows us to add custom code, where needed, at the outermost level of containership:

- *Accept* Instructs the socket to initiate communications with the specified client. Code in this method passes the specified request ID to the Winsock control's *Accept()* method.
- *Close* The Winsock control's native *Close()* method has been modified to call this container template method whenever a client socket closes the connection.
- *ConnectionRequest* The Winsock control's native *ConnectioRequest()* method has been modified to call this container template method, passing the incoming Request ID, when a request for connection is received.
- *DataArrival* The Winsock control's native *DataArrival()* method has been modified to call this container template method, passing the number of bytes in the incoming data stream when data is detected.
- *SendComplete* The Winsock control's native *SendComplete()* method has been modified to call this container template method when a client sends a "completed" signal.

The xWSListener class

The listener's function is to detect, and initiate the response to, a client's request for a connection. When a request is detected, the socket's *ConnectionRequest()* event fires and passes the request ID to the associated method. This, as described previously, is passed on to the container's *ConnectionRequest()* method. In the Listener subclass, all this method does is pass the request on to its parent's *ConnectTo()* method:

```
*** ActiveX Method, surfaced in container
LPARAMETERS tnRequestID
*** Pass connection Request to Parent to deal with
IF PEMSTATUS( This.Parent, "ConnectTo", 5 )
  This.Parent.ConnectTo( tnRequestID )
ENDIF
```

A custom *Listen* method has been added to the container, which surfaces the socket's native *Listen()* method in the container. The code is:

```
*** Pass call on to socket object
This.oSocket.Listen()
```

Finally, a custom property named *LocalPort* has been added to surface the socket's native property of the same name in the listener class container. It has been given an access method:

```
*** Simply return the current setting from the socket
RETURN This.oSocket.LocalPort
```

and an assign method:

```
LPARAMETERS tnPort
IF VARTYPE( tnPort ) = "N" AND ! EMPTY( tnPort )
  This.oSocket.LocalPort = tnPort
ENDIF
```

As you can see, we never actually use the container level property at all; the access and assign methods are used to re-direct all interaction to the socket's native *LocalPort* property instead. (Note: If you use this technique when dealing with the properties of contained objects, you may prefer to modify the methods so that the container level property is always updated. The reason is that otherwise it only shows its default value in the debugger, not the value of the underlying property.)

The xWSLogfile class

The second subclass is specialized to handle the task of receiving data from a client and writing that data out to a file. It has a number of custom properties, as listed in **Table 13**.

***Table 13**. Custom properties for the xWSLogfile class.*

Property	Description
cFileName	Name of the current output file. Generated when the first packet of data is received. Subsequent packets are added to the end of the current file.
cInstanceName	Unique instance name for the socket. Generated and passed to the object's *Init()* method when the object is created. Used to match an instance of the socket to its entry in the server's sockets collection.
oParent	Reference to the owning server object. Passed to the object's *Init()* method when the object is created. Used to initiate a request to the server to release the socket when its connection is closed.
State	Exposes the socket's native State property as a read-only property at the container level. (Access and assign methods make the property read only.)

Apart from the *Init()* method, which simply transfers the passed in parameters to the relevant properties, and the access and assign methods, which make the *State* property behave as if it were read only, there are only two methods that contain any code.

The *Close()* method is called from the socket's native *Close()* event, which is fired when the connection is closed. The code here uses the stored reference to the server object to initiate its own suicide, but first ensures that the garbage is collected by clearing the property that holds the reference to the server:

```
*** Called when connection is closed
*** Get a ref to the parent
loParent = This.oParent
*** Collect the garbage!
This.oParent = NULL
loParent.RemoveSocket( This.cInstanceName )
```

The *DataArrival()* method contains the specific code that writes the data sent by the client out to a file. The method receives, as a parameter, the number of bytes in the current transmission:

```
LPARAMETERS tnDataLen
LOCAL lcFileName, lcData
WITH This
  IF EMPTY( tnDataLen )
    *** NO data - do nothing
    RETURN
  ENDIF
```

We then check the custom *cFileName* property. This will be empty unless the current data packet is a continuation of a previous transmission. Remember, one of the benefits of using TCP is that we are not limited to a single send operation, but it means that we need to design for the possibility of multiple packets of data being transmitted:

```
  *** Get a filename if necessary
  IF EMPTY( This.cFileName )
    *** Create a new file (remove [] first!)
    lcFileName = .cInstanceName + TTOC( DATETIME(), 1 ) + ".txt"
    .cFileName = lcFileName
  ELSE
    *** This is a continuation data stream
    lcFileName = ALLTRIM( .cFilename )
  ENDIF
```

All that is left to do is to initialize the buffer to the correct length and call the socket's *GetData()* method to read the data in and then use **STRTOFILE()** to add the data stream to the output file:

```
  *** Initialize the buffer
  lcData = REPLICATE( " ", tnDataLen )
  *** And read the data stream into the buffer
  .oSocket.GetData( @lcData, "" , tnDataLen )
  *** Create the File, adding this data block
  *** to the end if the file is already there
  STRTOFILE( lcData, lcFileName, 1 )
  RETURN
ENDWITH
```

That is all that is required to actually receive a data log and write it out to file. The only real complexity is in the server, where we need to manage the sockets collection.

The xTCPServer class

The server class, as described earlier, is actually based on a form class onto which an instance of the Listener class has been placed. The server's *Init()* method includes code to accept, and set, a specific port for the listener before calling its *Listen()* method.

```
LPARAMETERS tnLocalPort
WITH This
  *** Use a specific port if passed in, otherwise leave
  *** at whatever the class defiens as default
  IF ! EMPTY( tnLocalPort )
```

```
    .oListener.LocalPort = tnLocalPort
  ENDIF
  .oListener.Listen()
ENDWITH
```

Code has also been added to the server's *Destroy()* method to ensure that it first removes any sockets and then explicitly releases the listener before allowing itself to be released.

It has a custom *cSocketClass* property, which is used for the name of the subclass that is to be instantiated when a connection is required. Two custom methods, *RemoveSocket()* and *ConnectTo()*, manage the sockets collection, which, apart from hosting the listener, is the main function of the server object.

The *RemoveSocket()* method is called from the *Close()* method of socket when its connection is closed. The socket passes its own instance name as a parameter, which is used as an index into the sockets collection. Having found the right socket, it is released and the collection's counter is updated and the array re-dimensioned, as follows:

```
LPARAMETERS tcName
LOCAL lnItem
lnItem = 0
WITH This
  *** Get the row number for this socket from the array
  lnItem = ASCAN( .aSockets, tcName, -1, -1, 1, 15 )
  IF lnItem > 0
    *** Found it, get an object reference
    loSocket = .aSockets[ lnItem,2 ]
    *** And release it
    loSocket.Release()
    *** Remove the element from the array
    .nSockets = IIF(.nSockets > 1, .nSockets - 1, 1 )
    ADEL( .aSockets, lnItem )
    *** Re-Dimension the array
    DIMENSION .aSockets[ .nSockets, 2 ]
  ENDIF
ENDWITH
```

The *ConnectTo()* method is called, with a numeric Request ID, from the Listener's *ConnectionRequest()* method. It initiates the call to a series of protected methods on the server, which return a reference to an available socket in the sockets collection. The Request ID is then passed to the *Accept()* method of the specified socket. (Note that in this, very simple, example there is no real error handling, which, for a full production implementation, should be added to this method.)

```
LPARAMETERS tnRequestID
LOCAL llRetVal
*** Get a Reference to an open Socket Object
loRef = This.GetSocket()
*** If this is not a valid object, bale out
llRetVal = ( VARTYPE( loRef ) = "O" )
IF llRetVal
  *** Tell it to connect
  loRef.Accept( tnRequestID )
ELSE
```

```
  *** Add proper error handling here
  MESSAGEBOX( "Unable to get a socket", 16, "Connection Failed" )
ENDIF
RETURN llRetVal
```

The code in the protected *GetSocket()*, *PollSockets()*, and *AddSocket()* methods is merely standard Visual FoxPro code to either find an existing socket that is free, or create a new socket, and return an object reference. However, as you may have noticed, the design of this example is such that when a socket's collection is closed, the socket is simply released. So why bother?

The reason is that while this particular example was designed with error logging in mind and therefore would not (we hope!) be dealing with large numbers of transactions, the server class can instantiate any subclass (just change the *cSocketClass* property) and it would not always be best to release a socket just because its connection has been closed. In fact, in the interest of performance, new sockets should only be created when no existing sockets are free.

Implementing the error logger

To run this sample on your local machine, you will need to carry out the following steps:

1. In the current instance of Visual FoxPro, set the default directory to the location that contains CH09.VCX. Create an instance of the server object as follows:

   ```
   SET CLASSLIB TO ch09.vcx
   oLogger = CREATEOBJECT( 'xtcpserver' )
   ```

2. Create a second instance of Visual FoxPro and set the default directory to the location that contains the program TCPCLIENT.PRG. Instantiate the client object and then call its *PostData()* method passing the text you want to send, as follows:

   ```
   loClient = NEWOBJECT("xTCPClient","tcpclient.prg")
   loClient.PostData( "This is a simple test message" )
   ```

3. In the first instance of VFP, open the text file that was created when the message was received. Remember, it will be named with the instance name of the socket plus the date and time it was created and will, therefore, be something like this: "NW0GAG5Y20020506073602.TXT".

The implementation over a network is very simple indeed. Simply create an instance of the server class on the machine where error logs are to be collected:

```
SET CLASSLIB TO ch09.vcx
oLogger = CREATEOBJECT( 'xtcpserver' )
```

On the client machines, you either instantiate the client class as a global object in your application, or just when needed. (For error reporting, our personal preference would be to have the object available rather than having to instantiate it because who knows how stable the system is at that point?) Either way, all that is necessary is to collate the contents of your error report into a text string and pass it to the client's *PostData()* method.

The following code gets the contents of memory into a string, connects to a server named "*acs-server*" and transmits the memory dump.

```
*** Get contents of memory (excluding system bvars) into a string
LIST MEMORY LIKE * TO FILE dumpmem.txt NOCONSOLE
lcErrorText = FILETOSTR( 'dumpmem.txt' )
*** Create the client object
oErrCli = NEWOBJECT( 'xTCPClient', 'tcpclient.prg', NULL, 'acs-server' )
*** Pass the content as a string
llOk = oErrCli.PostData( lcErrorText )
IF llOK
  *** Message was sent
  *** Remove the local file
  DELETE FILE dumpmem.txt
ELSE
  *** Could not connect - do something appropriate
  *** Display a message, create a local log file, or whatever!
ENDIF
```

Winsock control—conclusion

While very simplistic, we hope that this example will give you the confidence to dig deeper into the possibilities offered by the Winsock control. It is a very powerful and flexible tool that can be used for much more than simply logging errors and exchanging messages across your local area network.

ActiveX controls, the last word

We hope that this (rather lengthy) chapter has helped to de-mystify the intricacies of working with the most useful ActiveX controls that are available to you. There are, of course, many more controls, some available as freeware, shareware, and commercial products. Whatever their source, they all have one thing in common. They are designed to make functionality available with the bare minimum of instance-level code and, by using them properly, you can often make your own life as a developer much simpler.

Chapter 10
Putting Windows to Work

The operating system offers us a rich selection of tools that we can use to tap into its functionality from our Visual FoxPro applications. Perhaps the most obvious of these is the Windows API. But we are not limited to only the WinAPI. The Windows Script Host is a language-independent scripting engine that allows us, among other things, to do batch processing. In this chapter, we explore the ways in which we can put Windows to work for us in our applications. Many thanks to George Tasker for his loader script and for his assistance with the Windows Script Host.

How do I work with the Windows Registry?

Before we dive into the details of *how*, a few words about what the Registry is and how it is organized may be helpful. The Windows Registry is a hierarchical database that is tightly integrated with the operating system. This means that its contents are always available to any application without needing to worry about setting paths or changing directories. The operating system exposes the contents of the Registry through a set of functions that are defined as part of the Windows Application Programming Interface (API). In order to read from, and write to, the Registry these functions have to be used.

The structure of the Registry

The Registry is a hierarchical collection of *Keys*, *Values*, and *Data*. Unfortunately, the nomenclature chosen by Microsoft is not very user-friendly. Information stored in the Registry is held in Value/Data pairs, where the *Value* is actually the property or item name that is associated with the information saved as the *Data*. A *Key* is an identifier that groups one or more values and their associated data. Keys define the hierarchy that always starts from one of the predefined *Root Keys* and is built by adding a series of *Sub-keys* that define logical groupings of values. The *Data* for a value is stored in one of three basic formats:

- Null-Terminated String for character data. Defined as Type = 1 (*REG_SZ*)
- Double Word (4 byte) for integer data. Defined as Type = 4 (*REG_DWORD*)
- Binary for all other data. Defined as Type = 8 (*REG_BINARY*)

Note that we only need to worry about the first two of these data types (character and integer) because, in Visual FoxPro, we cannot work directly with the binary data type.

If you use the editing functions provided in the Windows Registry Editor, you will find that the various types of Registry entries are always referred to in the dialogs as either *Key*, *Value*, or *Data*. However, in the main display, the *Value* column is, for some reason that is beyond our comprehension, titled *Name*.

Figure 1 shows the Windows Registry Editor tool opened to display the current user's color settings. Notice how the display reflects the way in which the information is organized. The left hand panel shows the Keys (starting from the root keys) and their hierarchy of sub-

keys. The right hand panel lists, for the currently selected key, the values and their associated data types and data.

One useful function to remember is that the Edit pad of the menu includes an option to copy the currently selected key (as shown in the bottom left corner of the window) to the clipboard.

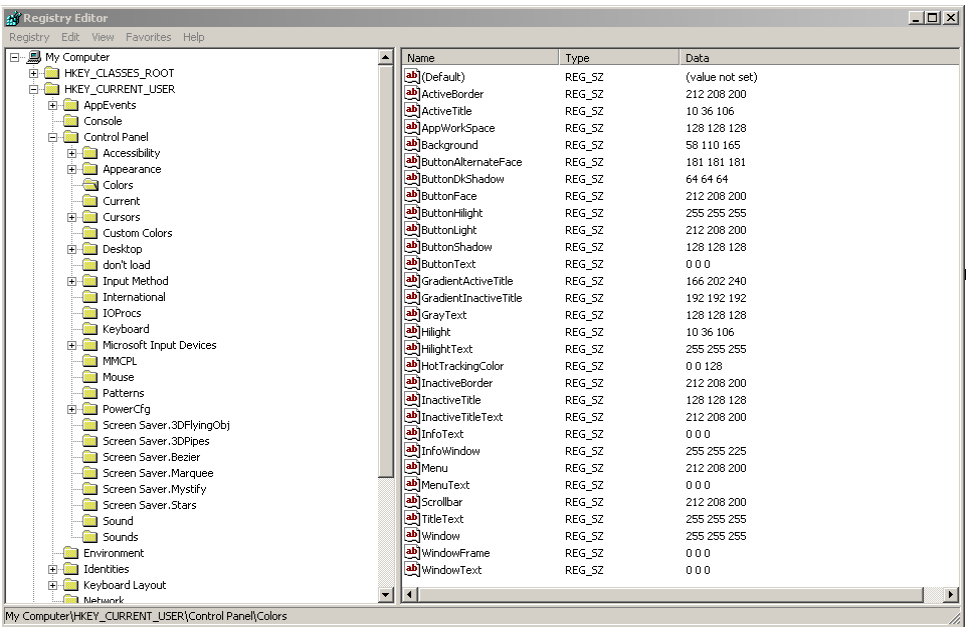


***Figure 1**. The Windows Registry structure.*

Depending upon the version of Windows, the Registry has either five (Windows NT, Windows 2000, and Windows XP) or six (Windows 95 and 98) pre-defined "Root" keys. As noted earlier, all Registry keys must, ultimately, descend from one of these. Each root key is identified by a numeric value (or *handle*) as shown in **Table 1**.

***Table 1**. Registry root keys.*

Name	Handle
HKEY_CLASSES_ROOT	-2147483648 = BITSET(0,31) [0x80000000]
HKEY_CURRENT_USER	-2147483647 = BITSET(0,31)+1 [0x80000001]
HKEY_LOCAL_MACHINE	-2147483646 = BITSET(0,31)+2 [0x80000002]
HKEY_USERS	-2147483645 = BITSET(0,31)+3 [0x80000003]
HKEY_CURRENT_CONFIG	-2147483643 = BITSET(0,31)+5 [0x80000005]

The sixth key, used only in Windows 95 and 98, was used to store certain system configuration information in RAM. It was re-created every time the system booted and is not used in later versions of Windows (and it is not really relevant in an application context anyway). The five main keys are used as follows:

- HKEY_CLASSES_ROOT — File extension associations and COM class registration information.
- HKEY_CURRENT_USER — User profile for the currently logged in user. A new HKEY_CURRENT_USER structure is created each time a user logs on to a machine.
- HKEY_LOCAL_MACHINE — Information about the local computer system, including hardware and operating system data such as bus type, system memory, device drivers, and startup control parameters.
- HKEY_USERS — All defined user profiles. Profiles include environment variables, personal program groups, desktop settings, network connections, printers, and application preferences.
- HKEY_CURRENT_CONFIG — Configuration data for the current hardware profile.

Full details of how the Registry is structured, and the contents of the major sub-keys, can be found in the *Windows Resource Kit Reference*, which is available through MSDN.

So, when should I be using the Registry?

There are some very good reasons why you might need to work directly with the Registry in a Visual FoxPro application. The first, and most obvious, is to get access to information that either Windows itself, or other applications, have stored about the machine on which your application is running. For example, setting up to work with e-mail, or with remote data, will almost inevitably require retrieving information about installed software or components from the Registry.

A second good reason is so that your application can restore, and save, an individual user's configuration and/or preferences. The days when we could simply impose our own standards for the look and feel of an application upon users have long gone. Not only are users more sophisticated generally, but they are used to being able to configure applications to look and run in the way in which they like. The Registry is specifically designed for handling this issue, and all that is required is to read and write settings in the "*Current User*" branch to have them associated directly with the individual user.

Finally, the Registry is a good choice when you need to store specific information, such as registration data, on each machine on which an application has been installed. By writing this information into the "*Local Machine*" branch of the Registry tree, you ensure that it is associated with the machine rather than any specific user. You also gain a degree of protection for sensitive information because it is less easy for the casual user to find, or make changes to, values that have been stored in the Registry.

Of course, using the Registry for more general application-specific information can be a double-edged sword. There may well be occasions when you would *want* an end user to be able to modify such information, and the Registry is not the most user-friendly environment for the uninitiated. As a general rule we would advocate keeping purely application-specific

data in an alternative format (either a FoxPro table, XML file, INI file, or just plain ASCII text) so that it can be stored directly with, and managed as a part of, the application itself.

How do I access the Registry?

As mentioned in the introduction to this section, the Windows API includes a set of functions that deal specifically with the Registry. You can either just declare and use the functions directly in your code, or as a much better solution use a class that provides a wrapper around the functions and hides much of the complexity. The sample code includes such a class (x*RegBase*) and a specialized subclass for reading and writing to the Visual FoxPro Options key (*xFoxReg*).

But isn't there already a Registry class in the FFC?

Ah, yes, so there is. The class library is named REGISTRY.VCX, and it contains a root class that provides basic list, get, and set functions for keys, plus a number of specialized subclasses (see **Table 2**).

***Table 2**. Classes in FFC registry.vcx.*

Class	Description
registry	Provides access to information in the Windows Registry.
foxreg	Specialized subclass of "registry" to access VFP options.
filereg	Specialized subclass of "registry" to access information about applications.
odbcreg	Specialized subclass of "registry" to access information about installed ODBC drivers.
oldinireg	Specialized subclass of "registry" to access Windows INI files.

However, the Registry class in the FFC has a couple of shortcomings that, in our opinion, make it virtually useless. First, its *get* and *set* methods can only handle character data, which is fine for Visual FoxPro because even its "numeric" information is stored as strings. However, that is not generally true for other applications and it is a serious limitation. Second, the class is designed as a visual class, and includes code that calls the `MessageBox()` function when an error occurs. This means that it cannot be used in the middle tier of an application or in a COM component. Third, the actual code is old and has not been updated since the introduction of Windows 95 (check out the *Init()* method). Finally, it is (as is usual in the FFC, alas) neither well commented nor properly documented. Our class addresses all of these issues and exposes the same public interface as the FFC *registry* class, so that it is interchangeable with it.

Structure of xRegBase class (*Example: mfRegistry.prg*)

The class is actually based on the custom *xObjBase* class (which inherits from the native "custom" base class through our generic *xCus* subclass). This class provides a standard error logging mechanism and adds a couple of simple custom methods. It is our standard root class for objects that have no user interface and that are defined in code.

In order to actually read from, or write to, the Registry we need to get a handle to the *key* that owns the *value* whose *data* we want to access. This handle is returned when we "*open*" the key, and we have defined a custom property (*nCurrentKey*) that is used to store it. However, before we can get a handle to a key, we must also know which of the five root keys is its ultimate owner. In order to avoid the necessity of passing the root key handle explicitly to

every function call, we have defined a custom property (*nCurrentRoot*) to store it. A custom method (*SetRootKey()*) uses simple integers to identify and set the root key handle. By default the class sets the current root key to HKEY_CURRENT_USER since this is the most usual setting required. The full set of properties and methods is shown in **Table 3**.

***Table 3**. Properties and methods of the xRegBase class.*

Property	Description
nCurrentRoot	The handle of the current Registry root key. Defaulted to HKEY_CURRENT_USER.
nCurrentKey	The handle of the currently open sub-key. Defaulted to 0.
lDoneDLLs	Flag set after API functions have been declared to prevent repeated declarations.
lCreateKey	Flag to control auto-creating keys from "Open". Defaulted to False.
Method	**Description**
ChkVersion (Protected)	Called from *SetRootKey()* to check platform and actually set the root key property if running under Windows.
CleanKey (Protected)	Removes leading and trailing path separators from a key string.
CloseKey (Protected)	Closes the key pointed to by the *nCurrentKey* property.
CreateKey (Protected)	Called from *SetRegKey()* when a key is not found and needs to be created. Works through the key string and creates all necessary sub-keys.
DeleteKey	Deletes the specified item (either "value" or sub-key) and all child items.
Destroy	On destroy, releases the DLLs opened by *LoadAPICalls()*.
FoxToReg (Protected)	Returns a VFP value (String or Integer) as a Registry value (REG_SZ or REG_DWORD).
GetKeyValue (Protected)	Returns the "data" associated with the specified "value".
GetRegKey	Returns the content of the "data" property for the specified "value" in the defined sub-key.
Init	On initialization calls *SetRootKey()*.
IsKey	Returns True if the specified Registry handle contains the named sub-key.
ListKeyNames (Protected)	Populates the named array (passed by reference) with the list of "sub-keys" for the current key.
ListKeyValues (Protected)	Populates the named array (passed by reference) with both "values" and "data" for the current key.
ListOptions	Populates the named array (passed by reference) with either the values alone, or both values and data, for the defined sub-key.
LoadAPICalls (Protected)	Declares the API functions that are called later by other methods as needed.
OpenKey (Protected)	Attempts to open the specified sub-key, returns a numeric handle to the key if successful. If passed a parameter, or *lCreateKey* property is set, will attempt to create the key if it does not already exist.
RegToFox (Protected)	Returns a Registry value (REG_SZ or REG_DWORD) as a VFP string or integer.
SetKeyValue (Protected)	Sets the "data" property of the specified "value".
SetRegKey	Sets the "data" property for the specified "value" in the defined sub-key.
SetRootKey	Sets the *nCurrentRoot* property according to passed in key number: 1 = HKEY_CURRENT_USER 2 = HKEY_USERS 3 = HKEY_LOCAL_MACHINE 4 = HKEY_CLASSES_ROOT 5 = HKEY_CURRENT_CONFIG

How do I read data from the Registry? (*Example: frmViewReg.scx*)

The only tricky part of this is ensuring that the required value is specified correctly. The API functions that access the Registry are constructed in such a way that in order to access data, you must first *open* the key that owns the value whose data you want. Opening a key returns a numeric *handle*, which must then be passed explicitly to the function that returns the data. To make things a little more complex, in order to actually open a key you must pass, by value, the handle for the root key that is its ultimate parent and the full path from the root key to the required key as a character string. So, if we wanted to read the setting of one of the values in the current user's color preferences, which are stored in the following key:

```
HKEY_CURRENT_USER\Control Panel\Colors
```

we would need to open the key by calling the appropriate API function and passing the key name as follows:

```
Control Panel\Colors
```

Notice that when passing the path we must omit the leading "\" character, and its root key, separately, as:

```
-2147483647
```

The *GetRegKey()* method in *xRegBase* hides all the complexity associated with this process and allows you to retrieve a value directly by passing the following parameters:

- The name of the value whose data is required.
- The relative path to the key which owns the required value.
- The handle of the key that owns the specified key. When not passed explicitly, this value defaults to whatever is set as the current root key in the object.

The reason that this method is constructed in this fashion is so that we can pass the full path from the owning root key directly without having to descend one level at a time, opening each key in turn. The following code snippet shows how this method can be used to retrieve the current user's color setting for highlighted text:

```
*** Instantiate the class
SET PROCEDURE TO mfregistry
oReg = CREATEOBJECT( 'xRegBase' )
*** We want the value named "Hilight" from the User's color settings
lcKey = oreg.Getregkey( 'hilight', '\Control Panel\Colors' )
*** The return is a space-separated string for RGB settings, so convert it
lnColor = EVALUATE("RGB(" + CHRTRAN( lcKey, " ", "," ) + ")")
*** Show the color number
? lncolor
```

When using the GetRegKey() method to retrieve default values from the Registry, you must pass an empty string as the first argument. Although the Registry Editor displays the item's name as "(Default)", there is, in fact, no item name present in the Registry.

The class also includes a *ListOptions()* method that will allow you to retrieve, into an array, either the list of sub-keys for a key, or the list of values and their associated data. The method takes four parameters as follows:

- The array to be populated with results. Must be passed by reference.
- The relative path of the key whose child values are required.
- The logical value to return only sub-keys. Default behavior is to return both Values and Data.
- The handle of the key that owns the specified sub-key. There are three ways of passing this parameter:
    - If it is omitted, whatever is defined as the currently open key is assumed to be the parent (if no key is open, the default root key is assumed).
    - If it is zero, any currently open key is first closed and the currently defined root key is assumed to be the parent.
    - If it is a non-zero numeric value, that value is assumed to be the handle to the parent of the specified key.

The following snippet shows how this can be done interactively, first to get all the sub-keys that are defined under the *Control Panel* key:

```
*** Instantiate the class
SET PROCEDURE TO mfregistry
oReg = CREATEOBJECT( 'xRegBase' )
*** Get the list of keys under the Control Panel key
DIMENSION laKeys[1]
llOk = oreg.ListOptions( @laKeys, '\Control Panel', .T. )
DISPLAY MEMORY LIKE laKeys*
```

and second, to return the actual values, and their data, for the *Colors* sub-key:

```
*** Instantiate the class
SET PROCEDURE TO mfregistry
oReg = CREATEOBJECT( 'xRegBase' )
*** Get the list of keys under the Control Panel key
DIMENSION laVals[1]
llOk = oreg.ListOptions( @laVals, '\Control Panel\Colors' )
DISPLAY MEMORY LIKE laVals*
```

Note: If you don't want to explicitly release and re-create instances of the Registry class for each key that you wish to interrogate, you must always pass the fourth (Parent Key)

parameter to *ListOptions()* explicitly. To write the results from the preceding two snippets to a text file, the code would have to be amended to look like this:

```
*** Delete the output file if it exists
IF FILE( "regvals.txt" )
  DELETE FILE regvals.txt
ENDIF
*** Instantiate the class
SET PROCEDURE TO mfregistry
oReg = CREATEOBJECT( 'xRegBase' )
*** Get the list of keys under the Control Panel key
DIMENSION laKeys[1]
llOk = oreg.ListOptions( @laKeys, '\Control Panel', .T. )
*** DISPLAY MEMORY LIKE laKeys*
LIST MEMORY LIKE laKeys* TO FILE regvals.txt ADDITIVE NOCONSOLE
*** Get the list of keys under the Control Panel key
DIMENSION laVals[1]
llOk = oreg.ListOptions( @laVals, '\Control Panel\Colors', ,0 )
*** DISPLAY MEMORY LIKE laVals*
LIST MEMORY LIKE laVals* TO FILE regvals.txt ADDITIVE NOCONSOLE
MODIFY FILE regvals.txt NOWAIT
```

The example form included with this chapter (see **Figure 2**) illustrates a simple "Registry Viewer" that uses the *ListOptions()* method in two different ways.

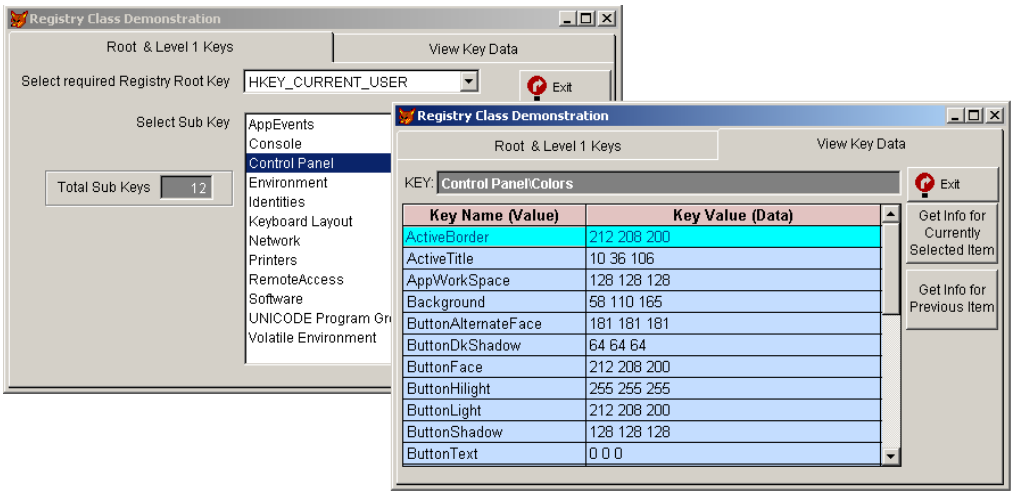


***Figure 2**. Simple VFP Registry Viewer (frmviewreg.scx).*

The first is shown by using the *GetKeyList()* method, which populates an array property on the form with the names of the sub-keys for the currently selected root key. This array is used to populate the list box on the first page of the form. The second is illustrated by using the *GetKeyVals()* method, which populates a cursor with the names of any sub-keys and any values (with their data) for the currently selected key. This cursor is used to populate the grid on the second page of the form.

How do I write data to the Registry? (*Example: WriteReg.prg*)

It makes little difference whether you are talking about updating existing entries or creating entirely new entries; the basic methodology is the same as for reading values. First you must open the owning key and then write the data to a specific value within that key. Of course, this immediately raises the question of what to do if the key whose value you are trying to set does not exist. This is all handled transparently by the *SetRegKey()* method, which accepts the following parameters:

- The *Value* (item name) for which *data* is to be created or updated.
- The *Data* to be written to the specified value.
- The relative path of the key that owns the specified value.
- A flag that, when set, overrides the setting of the *lCreateKey* property to allow keys to be created if they do not exist.
- The handle of the key that owns the specified key. When not passed explicitly, this value defaults to whatever is set as the current root key in the object.

The sample program (WRITEREG.PRG) uses this method to create a set of Registry entries for a mythical application, consisting of a registration value under the "local machine" root and some default settings under the "current user" root. **Figure 3** shows the current user entries.

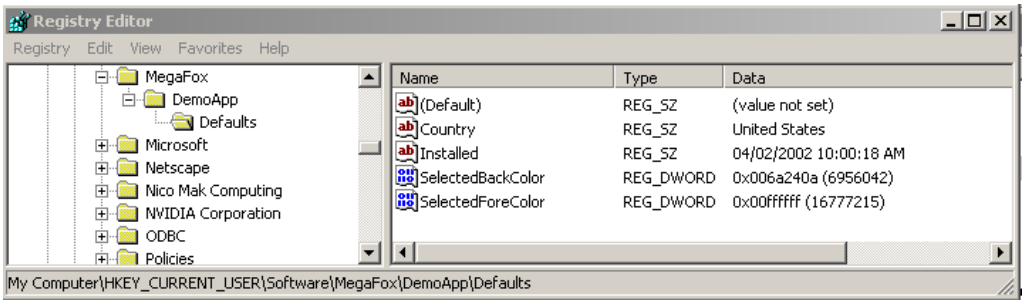


***Figure 3**. Creating current user keys.*

As you will see when you examine the sample program, using the *xRegBase* class makes creating and setting keys very simple indeed. There are only three steps:

1. Set the correct root key.
2. Set up variables for the required sub-key, the value, and its data.
3. Call *SetRegKey()* and check the return value.

How Registry keys are created

The process of creating Registry keys using the API functions is relatively simple but must be done one step at time. This is because the functions that actually create keys always need to be passed the handle to the immediate parent key. So in order to set a value for a key like the one specified in the sample code:

```
HKEY_LOCAL_MACHINE\SoftWare\MegaFox\DemoApp\Registration
```

we first need to get a handle to it. However, if the key does not exist, we cannot actually create it with a single function call. The analogy with directory structures fails at this point because it is perfectly possible to go to the DOS command line and use the "make directory" command like this:

```
MD TestDir\MegaFox\Working\DirTest
```

In order to create a key by passing the entire Registry path at once, we must determine the lowest point in that path that already exists within the Registry. This requires stepping back through the path, removing one key level at a time and trying to open the resulting key. Assuming that we find a key that we can open, we then start working forward again, adding the first "new" sub-key and opening it. This stepping process continues, using the handle to the last key created as the parent for the next until we have created the entire sequence of levels to support the required key.

We could, of course, deal with this explicitly in our code, by calling the *SetRegKey()* method repeatedly like this:

```
*** Create the registry object
oReg = NEWOBJECT( 'xRegBase', 'mfregistry.prg' )
WITH oReg
  *** Set Root key
  .SetRootKey( 2 )  && Local Machine
  *** Open/create the SoftWare sub-key
  .SetRegKey( "", "", "SoftWare", .T. )
  *** Now Open/create the MegaFox sub-key
  .SetRegKey( "", "", "MegaFox", .T. )
  *** Now Open/create the DemoApp sub-key
  .SetRegKey( "", "", "DemoApp", .T. )
  *** Now Open/create the Registration sub-key and set the value
  .SetRegKey( "MFDemo", "MFDEM-CH10S-WR1T5", "Registration", .T. )
ENDWITH
```

but that defeats the purpose of using the class whose main benefit is that it hides this sort of complexity. We can simply use code like this to accomplish the exact same thing:

```
  oReg.SetRegKey("MFDemo", "MFDEM-CH10S-WR1T5", "Registration", .T. )
```

In the class, the actual code for dealing with opening and creating keys is contained in the two protected methods named *OpenKey()* and *CreateKey()*.

Deleting Registry keys

Deleting Registry keys is, essentially, the reverse of the creation process. The *xRegBase* class includes a *DeleteKey()* method that can be used to delete any key, at any level of the hierarchy, and all of its associated data. However, because of the danger associated with deleting items from the Registry, this method will not delete any key that contains sub-keys.

Therefore, to remove a key, and all of its dependent sub-keys, we must first delete all the lowest level sub-keys and then work up the hierarchy, deleting all keys at the same level as we go. The code in DELETEREG.PRG illustrates how this can be done and can be used to remove all the keys that were created by the WRITEREG.PRG.

How do I change Visual FoxPro Registry settings? (*Example: xFoxReg*)

Like most Windows applications, Visual FoxPro stores a number of key settings in the Registry. The majority of these are kept in the *Options* key under a version-specific sub-key (for example, *6.0* or *7.0*) located in the user settings tree at "*\Software\Microsoft\VisualFoxPro*" and, while most of them can be accessed programmatically through the SET commands, changes made in that way do not persist between sessions.

For this reason we have used the VFP settings to illustrate how easily the generic *xRegBase* class described earlier can be subclassed to deal with a specific group of Registry keys. Here is the entire class definition:

```
DEFINE CLASS xfoxreg AS xRegBase
  *** This.cVFPOpt points to the VFP Key
  cVFPOpt = "Software\Microsoft\VisualFoxPro\" + _VFP.Version + "\Options"

  FUNCTION Init()
    *** Set up to use HKEY_CURRENT_USER
    This.SetRootKey( 1 )
  ENDFUNC

  FUNCTION SetFoxOption( tcItemName, tcItemValue )
    *** Set a specific FoxPro Options Item
    RETURN This.SetRegKey( tcItemName, tcItemValue, ;
                           This.cVFPOpt, This.nCurrentRoot )
  ENDFUNC

  FUNCTION GetFoxOption( tcItemName )
    *** Read an Item
    RETURN This.GetRegKey( tcItemName, This.cVFPOpt, ;
                           This.nCurrentRoot )
  ENDFUNC

  FUNCTION ListFoxOptions( taFoxOpts )
    *** Build an array of items (3rd param = Names Only!)
    RETURN This.ListOptions( @taFoxOpts, This.cVFPOpt, ;
                             .F., This.nCurrentRoot )
  ENDFUNC
ENDDEFINE
```

All we have done here is to add a custom property to store the required parent key (*cVFPOpt*) and added some simple methods that wrap calls to the methods defined in the parent class ("*xRegBase*"). In order to populate an array with all of the values (and their data) for the current version of VFP, all we need to do is:

```
oReg= NEWOBJECT( "xFoxReg", "mfRegistry.prg" )
DIMENSION laOpts[1]
oReg.listFoxOptions( @laOpts )
```

Similarly, to manipulate individual values we can just call the appropriate methods:

```
*** Show the current setting for the Bell
? oReg.GetFoxOption( 'Bell' )
*** And set it ON
oReg.SetFoxOption( 'Bell', 'ON')
? oReg.GetFoxOption( 'Bell' )
*** And OFF
oReg.SetFoxOption( 'Bell', 'OFF')
? oReg.GetFoxOption( 'Bell' )
```

Conclusion

In this section we have given a brief explanation of how the Windows Registry works and have described a generic class, and an example of a specialized subclass, for working with the Registry. If you have to manipulate the Registry in your applications, either to store your own data or to retrieve information already there, these classes will make your life much easier. Perhaps more importantly, by creating specialized subclasses along the lines we have illustrated, you can make your own life much simpler.

What is the Windows Script Host?

The Windows Script Host (WSH) is a tool that uses two built-in scripting engines, VBScript and JScript, to access objects in the Windows operating system, such as files, folders, and network items. Script files, whether written in VBScript or JScript, are plain text files with either a VBS or JS extension, respectively. They can be created and edited with any text editor, such as Visual Notepad. Detailed information on using both VBScript and JScript can be found in the Tools and Scripting section of the MSDN Platform SDK Documentation under "Scripting".

Once it is installed, the Windows Script Host can run any script just by double-clicking the script file's icon. The Windows Script Host loads the appropriate scripting engine, which then executes the commands contained in the script. Moreover, since the WSH is capable of creating anything that exposes itself as an OLE Automation server, you can use it to manipulate an out-of-process server like Microsoft Word or a custom in-process DLL.

It is the perfect tool for automating Windows tasks and can be used to do the sort of things that would have required a batch file in the old days of DOS. The Script Host uses one of two executable files to run scripts depending upon where they are to be implemented. WSCRIPT.EXE is used to run scripts as Windows applications, and CSCRIPT.EXE is used to run them as console applications in a DOS window.

Natively, the WSH consists of several files, each of which defines one or more component objects. Thus, the *Scripting.FileSystemObect* lives in SCRRUN.DLL, the regular expression parser in VBSCRIPT.DLL and the *WScript.Shell* and *WScript.Network* objects in WSHOM.OCX. We'll begin our discussion of the WSH with the *WScript* object.

The *WScript* object is the root object of the WSH object model hierarchy and is unique in that it never needs to be explicitly instantiated before invoking its properties and methods. It is simply available from within any script file that can then access the properties of the *WScript* object to become "self-aware". The *WScript* object also exposes *CreateObject()* and *GetObject()* methods that allow the script to launch and control other applications. Some of the most important properties of the *WScript* object are:

- *Arguments*: A collection of command line arguments passed to the script.
- *FullName*: Fully qualified path and file name of the host executable (either CSCRIPT.EXE or WSCRIPT.EXE).
- *ScriptFullName*: Fully qualified path and file name of the currently executing script.
- *Version*: The version of the Windows Script Host object.

The *WScript.Shell* object has methods to run and configure other applications, for creating desktop shortcuts and modifying the Registry. Some of its most important methods are:

- *Run* Launches the application name passed to it. When passed the name of a file with an application associated with it, opens the file using the appropriate application. This is much more flexible than simply using the Visual FoxPro `RUN` command because it can wait until the application is finished running before returning control to VFP.
- *AppActivate* Sets system focus to a window based on its title.
- *CreateShortcut* Creates desktop shortcuts to files or URLs.
- *RegWrite* Creates a new Registry key or writes a new value for an existing key.
- *RegRead* Reads the value of a Registry key.
- *RegDelete* Deletes a Registry key.
- *SendKeys* Sends keystrokes to the foreground application.

The *WScript.Network* object has methods to get information about, and modify, network configurations. It can be used to map network drives and install printers. Its most important methods are:

- *EnumNetworkDrives* Returns a drives object containing information to identify network drives connected to the user's computer.
- *MapNetworkDrive* Maps a network drive to a drive letter.
- *RemoveNetworkDrive* Disconnects the specified network drive.
- *EnumPrinterConnections* Returns an object containing information about the printers installed on the network.

- *AddPrinterConnection* — Maps a network printer to a local device name.
- *AddWindowsPrinterConnection* — Under Window NT and Windows 2000, attaches a remote printer without assigning a local port.
- *RemovePrinterConnection* — Releases a printer mapped to the user's machine.
- *SetDefaultPrinter* — Sets the default printer.

The *Scripting.FileSystemObject* has methods to perform disk input and output operations. These include reading, writing, and deleting both files and directories. It also has methods that return information about the drives available on the user's system. Visual FoxPro has some native capability in this area, and we can create our own functions to do more. Where appropriate it is better to do so and avoid incurring the performance penalty imposed by passing data back and forth across a COM interface. However, the *FileSystemObject* also provides us with functionality that we either cannot get at all, or can only get with great difficulty, from Visual FoxPro. As Visual FoxPro developers, these are the methods in which we are most interested:

- *CopyFolder* — Copies an entire folder including all of its files and subfolders.
- *DeleteFolder* — Deletes an entire folder including all of its files and subfolders.
- *MoveFolder* — Moves an entire folder including all of its files and subfolders to a new location. Can also be used to rename a folder.
- *GetSpecialFolder* — Returns a folder object reference to one of the following Windows folders depending on the parameter passed: 0-The Windows folder, 1-The Windows/System folder, 2- The Windows temporary folder.
- *DriveExists* — Returns true if the specified drive exists.
- *GetDrive* — Returns an object that contains information about the specified drive including the drive letter, drive type, file system, free space, share name, and volume name. The object also has an *IsReady* property that, as its name implies, tells you whether or not the drive is ready.

Where can I get the Windows Script Host?

The Windows Script Host version 1.0 is shipped as an optional component with Microsoft Windows 98 and NT Option Pack 4. Version 2.0 ships as part of Windows 2000 and Millennium Editions and is installed as a standard component of Internet Explorer versions 4 and 5. If you are running Windows 95, you can download the Windows Script Host from the Microsoft Windows Script Technologies Web site (http://msdn.microsoft.com/scripting),

provided that you have either OSR 2 of the operating system or Internet Explorer version 4 or later installed. You can also go to this Web site to upgrade your current scripting engines to the latest version, which (at the time of writing) is 5.6. You may be wondering why the version number went directly from 2.0 to 5.6. In previous releases of the WSH, there were discrepancies between the version numbers of its component files, and this file versioning issue was resolved by skipping some version numbers. To find out what version of the Windows Script Host is currently installed, just double-click on DISPLAYVERSION.VBS, included with the sample code for this chapter, in Windows Explorer.

The Windows Script Host can be dangerous in the hands of someone who has malicious intentions. Version 5.6 of the Windows Script Host employs a new security model to prevent this type of abuse. System administrators can enforce strict policies that determine which users have privileges to run scripts locally or remotely. If access to the WSH has been restricted, one of the following error messages may occur when an attempt is made to run a script:

- *Windows Script Host access is disabled on this machine. Contact your administrator for details.*
- *Initialization of the Windows Script Host failed.*
- *Execution of the Windows Script Host failed.*

How do I determine whether the Windows Script Host is installed?

Before we can tap into the power of the Windows Script Host, we must ensure that it is installed on the client machine. Of course, we could just try instantiating one of its component objects and let our program crash, but there are better ways of determining whether the WSH is present.

First, we can check the Registry for the key of the WSH component that we want to use. This code uses the Registry class discussed earlier in this chapter to verify that the *Wscript.Network* component is installed:

```
SET PROCEDURE to MFregistry.prg ADDITIVE
oReg=CREATEOBJECT( 'xRegBase' )

*** Set the root to HKEY_CLASSES_ROOT
oReg.SetRootKey( 4 )

*** See if the object is registered
llIsRegistered = oreg.IsKey( 'wscript.network' )
```

Second, we can make sure that the file is actually present. We can do this programmatically by retrieving its location from the Registry and using the **FILE()** function to verify its physical presence. The following code illustrates this technique:

```
SET PROCEDURE to MFregistry.prg ADDITIVE
oReg=CREATEOBJECT( 'xRegBase' )
```

```
*** Set the root to HKEY_CLASSES_ROOT
oReg.SetRootKey( 4 )

*** Retrieve the CLSID from HKEY_CLASSES_ROOT\WScript.Network\CLSID
lcClsID = oReg.GetRegKey( '', '\WScript.Network\CLSID' )
lcSubKey = '\clsid\' + lcClsID + '\InProcServer32\'

*** Retrieve the fully qualified path and file name for the object
lcFile = oReg.GetRegKey( '', lcSubKey )
llFileExists = FILE( lcfile )
```

How do I use the Windows Script Host to automatically update my application? (*Example: MyApp.vbs*)

The real problem here arises when an application is deployed to and run from users' local machines. While this has significant performance benefits over running the application directly from a server, it makes updating the application more difficult. The solution is to use the WSH to run a "*loader*" script that can check version information (for example, the date and time the local application was last modified against that of master copy on the network). If the application on the network drive is newer, the script merely copies it to the local drive before launching it.

This same methodology can be used to install new programs or update the runtime files on the local computer when a new version of Visual FoxPro or a service pack is released. We can even do this in "silent" mode using the setup program generated by the InstallShield Express version that ships with Visual FoxPro 7.0. In this case, all that needs to be done is to re-run the setup program using the /q or /qn command line parameters. One caveat here is that if anti-virus software is running (as it must be in this day and age), it may intervene and prevent the script from running until the user gives permission.

To automatically update an application called MyApp from the version on the network, just create a script called MYAPP.VBS using Visual Notepad or your favorite text editor. The loader program assumes that the loader script, the application, and the configuration file all have the same file stem. It also assumes that a text file with this stem and a ".sup" extension is updated on the remote drive whenever a new setup program is available. This text file is created on the local drive when the script runs the setup program and is used by the script to determine whether the setup program needs to be run.

The loader script first creates instances of the *FileSystemObject* and the *Shell*. It then initializes the variables required to locate the local and remote applications, the setup program, and the configuration file.

```
Dim oShell, oFSO, cExe, cLocal, cRemote, cSetup, cStem, oRemote, cParmeters

Set oFSO = CreateObject( "Scripting.FileSystemObject" )
Set oShell = CreateObject( "WScript.Shell" )
cParameters = " -cMyApp.Fpw"
cStem = "MyApp"
cExe = cStem & ".exe"
cSetup = cStem & ".sup"
cLocal = "C:\LocalDir\"
cRemote = "F:\MyNet\Homedir\"
```

Next, it checks to see if there's a text file on the network with the same file stem as the application and the extension ".sup" (short for "SetUP") that is more recent than the local one. This ".sup" file is merely a text file that tells the script that the setup program must be run when the version on the remote drive is newer than the version on the local drive. If *NewerFile()* returns true, the setup program on the remote drive is executed. This handles the situation where a service pack has been released or a new version of Visual FoxPro has been released.

```
If NewerFile( cLocal & cSetup, cRemote & cSetup ) Then
  RunSetup cLocal & cSetup, cRemote & cStem
Else
```

Note that the script assumes that the setup program on the remote machine is located in a subfolder under the "home directory" that has the same file stem as the application. So, in our example script, the remote home directory (the location of the exe on the remote machine) is F:\MyNet\HomeDir. The setup program, SETUP.EXE, must be located in F:\MyNet\HomeDir\MyApp. Although MYAPP.SUP must be created on the network whenever a new setup program is placed there, it is automatically created on the local machine when RunSetup executes.

The next thing is to make sure that the run-time library has been properly installed. If this test fails, the setup program is, once again, run to correct the problem. This covers the situation where a computer has been upgraded, but the necessary installation has not been performed.

```
  If Not IsInstalled() Then
    RunSetup cLocal & cSetup, cRemote & cStem
  Else
```

Finally, the date/time stamps of the remote and local copies are compared. If the remote copy is more recent than the local, it is copied over to the local drive prior to executing the application itself. If the remote copy is not more recent, then the existing local copy is simply executed. The executable is only copied from the remote drive to the local drive if the text file with the .sup extension on the remote machine is not newer than the one on the local machine. In this example, if F:\MyNet\HomeDir\MyApp.sup is newer than C:\LocalDir\MyApp.sup, the script attempts to run F:\MyNet\HomeDir\MyApp\Setup.exe.

```
    If NewerFile( cLocal & cExe, cRemote & cExe ) Then
      Set oRemote = oFSO.GetFile(cRemote & cExe)
      oRemote.Copy cLocal
    End If
    oShell.Run( cLocal & cExe & cParameters )
  End If
End If
```

The *NewerFile()* function returns true if the second file passed to it is newer than the first. This will be the case if the first file does not exist or the last modification date of the second file is more recent than that of the first file. It uses the *FileExists()* method of the *FileSystemObject* and the *DateLastModified* property of the file object to accomplish this.

```
Private Function NewerFile( tcLocalFile, tcRemoteFile )
  Dim oFSO, oLocal, oRemote
  Set oFSO = CreateObject( "Scripting.FileSystemObject" )
  If Not oFSO.FileExists( tcLocalFile ) Then
    NewerFile = True
  Else
    If Not oFSO.FileExists( tcRemoteFile ) Then
      NewerFile = False
    Else
      Set oLocal = oFSO.GetFile( tcLocalFile )
      Set oRemote = oFSO.Getfile( tcRemoteFile )
      NewerFile = ( oRemote.DateLastModified > oLocal.DateLastModified )
    End if
  End If
End Function
```

The *IsInstalled()* function reads the Registry to determine whether the Visual FoxPro runtime files are installed on the local machine. If the Registry key can't be found, the installation program must be run.

```
Private Function IsInstalled
  Dim oShell, cKey, cValue
  Set oShell = CreateObject( "WScript.Shell" )
  cKey = "HKCR\VisualFoxPro.Runtime\CLSID\"
  On Error Resume Next
  cValue = oShell.RegRead( cKey )
  IsInstalled = ( cValue > "" )
End Function
```

The *RunSetup()* routine, as its name implies, installs the application on the local machine. It also creates the text file with the .sup extension so that it is possible to determine when the setup program needs to be re-run in the future. This will be whenever the MYAPP.SUP file on the network is newer than the local version.

```
Private Sub RunSetup( tcTextFile, tcRemotePath )
  Dim oFSO, oShell, oTest
  Set oFSO = CreateObject( "Scripting.FileSystemObject" )
  Set oShell = CreateObject( "WScript.Shell" )
  Set oText = oFSO.CreateTextFile( tcTextFile, True)
  oText.Close
  oShell.Run( tcRemotePath & "\Setup.exe /Q" )
End Sub
```

To use the loader script, amend the application startup shortcut so that it is pointing to the loader script instead of the application itself. That's all it takes to make sure that all the users are running the most current version of your application.

How do I use the Windows Script Host to read the Registry?

The *WScript.Shell* object has methods that enable to you read, write, and delete Registry entries. Although it is possible to implement this functionality in Visual FoxPro using the WinAPI, as we have seen, the code is voluminous. Getting individual key values from the Registry is a snap using the Windows Script Host.

One nice feature about using the *RegRead()* method is that you can abbreviate the Registry roots and you do not need the "magic" numbers that are required when using the API. In the code snippet that follows, HKCU is the abbreviation for HKEY_CURRENT_USER. The other valid abbreviations are:

- HKLM HKEY_LOCAL_MACHINE
- HKCR HKEY_CLASSES_ROOT
- HKEY_USERS HKEY_USERS
- HKEY_CURRENT_CONFIG HKEY_CURRENT_CONFIG

The Windows Script Host recognizes these abbreviations so there is no need to **#DEFINE** them. Another benefit is that it requires only a single method call to retrieve specific data. For example, if we want to highlight the current row in a grid using the colors that the user has set up in the Control Panel, this is all we need to do to get that information using the Windows Script Host:

```
loShell = CreateObject( 'WScript.Shell' )
lcBgColor = loShell.RegRead( 'HKCU\Control Panel\Colors\Hilight' )
```

This returns the RGB values of the highlight color as a space-delimited set of values. In order to use this to set the background color for the current grid row, call this code from the grid's *Init()*:

```
lcBgColor = 'RGB( ' + STRTRAN( lcBgColor, ' ', ', ' ) + ' )'
lcNormalBg = loShell.RegRead( 'HKCU\Control Panel\Colors\Window' )
lcNormalBg = 'RGB( ' + STRTRAN( lcNormalBg, ' ', ', ' ) + ' )'
This.SetAll( 'DynamicBackColor', ;
  "IIF( RECNO( This.RecordSource ) = This.nRecNo, " + ;
     lcBgColor + ", " + lcNormalBg + " )", 'COLUMN' )
```

We can use similar code to retrieve the user's setting for the color of highlighted and normal text from the *Window* and *HilightText* values, respectively.

How do I use the Windows Script Host to write to the Registry?

As we saw earlier in this chapter, using the WinAPI to write to the Registry poses problems if the parent keys do not yet exist. We needed an awful lot of code, and some fairly complex logic, in our custom *xRegBase* class to handle this situation seamlessly. Not so when we use the *RegWrite()* method of *WScript.Shell* to perform the same task. All it takes is a single method call.

Using our previous example of writing a registration key like this:

```
HKEY_LOCAL_MACHINE\SoftWare\MegaFox\DemoApp\Registration
```

to the Registry where we do not already have the MegaFox and DemoApp keys is trivial when the Windows Script Host is used to do it:

```
loShell = CreateObject( 'wscript.shell' )
loShell.RegWrite( 'HKLM\SoftWare\MegaFox\DemoApp\Registration', ;
                  'MFDEM-CH10S-WR1T5' )
```

How do I let the user choose which printer to use? (*Example: PrintDemo.scx*)

The good news is that we can make use of the *Wscript.Network* object's *SetDefaultPrinter()* method to temporarily change the setting of the default printer. The bad news is that the Windows Script Host has no method to retrieve the current default printer. Fortunately, we can use Visual FoxPro's `SET("PRINTER", 2)` function to return this information. We can also use the `APRINTERS()` function to obtain a list of installed printers to present to the user so that they can choose. The example form (see **Figure 4**) illustrates this process.

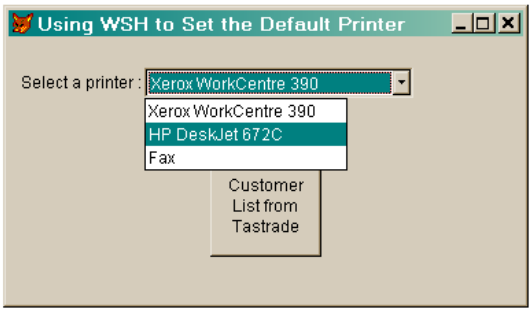


***Figure 4**. Using WScript.Network to set the default printer.*

This code, in the combo box's *Init()* method, sets up its *RowSource* to display the installed printers:

```
APRINTERS( This.aContents )
This.Requery()
```

Then we save the current default printer and instantiate *WScript.Network* in the form's *Init():*

```
WITH ThisForm
  *** Save the default printer
  .cDefaultprinter = SET( 'PRINTER', 2 )
  *** Set the combo up to point at the current default printer
  .cboPrinters.ListIndex = ASCAN( .cboPrinters.aContents, .cDefaultPrinter, ;
                                  -1, -1, 1, 15 )
  *** Create the WScript.Network object
  .oNet = CREATEOBJECT( 'WScript.Network' )
ENDWITH
```

Finally, this line of code, in the combo's *Valid(),* sets the default printer to whatever is selected by the user:

```
Thisform.oNet.SetDefaultPrinter( This.DisplayValue )
```

Another single line of code in the form's *Destroy()* method restores the default printer setting to whatever it was when the form was instantiated:

```
This.oNet.SetDefaultPrinter( Thisform.cDefaultprinter )
```

Although we can use the native **SET PRINTER TO** command to change the Visual FoxPro default printer, this may not be enough in some circumstances. For example, suppose we need to print a document using Word Automation, selecting the printer for the output. In this case, **SET PRINTER TO** does not do what we need, but the Windows Script Host does.

How do I delete an entire folder?

It takes a lot of code to do this in Visual FoxPro because the **RMDIR** command will only delete empty directories. This means that we need to use the **ADIR()** function to build a list of subfolders and then write code to drill down and delete all files before deleting each subfolder in turn. The *FileSystemObject* can do the exact same thing using a single method call and duplicates the functionality of the old DOS **DELTREE** command:

```
oFSO = CreateObject( 'Scripting.FileSystemObject' )
oFSO.DeleteFolder( 'MyFolder2Delete', .T. )
```

The second parameter tells the Windows Script Host to delete folders with the read-only attribute set. But do be careful when you use this method! The folders and files are deleted without being sent to the recycle bin, so once deleted they are gone forever.

How do I rename a directory?

Although we can use the native Visual FoxPro **RENAME <old file> TO <new file>** command to rename files, there is no equivalent command to rename directories. We do, however, have access to the *FileSystemObject* and are able to do this with very little code like this:

```
oFSO = CreateObject( 'Scripting.FileSystemObject' )
oFSO.MoveFolder( 'D:\MyOldFolder', 'D:\MyNewFolder' )
```

As you can see, the *MoveFolder()* method merely renames the folder if the destination folder is at the same level of hierarchy on the same drive. An alternative to using the *MoveFolder()* method is to merely change the folder's *Name* property like this:

```
oFldr = oFSO.GetFolder( 'D:\MyOldFolder' )
oFldr.Name = 'MyNewFolder'
```

The only downside to the latter is that it requires one more line of code, and we believe that less code is better because less code means fewer bugs.

How do I know whether a drive is ready?

The *FileSystemObject* has a *Drives* collection. Each object in the collection contains information about a specific drive on the system so it is very easy to get whatever information

is available for a drive. For example, if we want to know whether or not a specific drive exists, all it takes is a single line of code:

```
llDriveExists = oFSO.DriveExists( 'A' )
```

If the drive does exist, the next question we probably need to answer is "Is there a disk in it?" We can do this easily using the *IsReady* property of the drive object.

```
oDrive = oFSO.GetDrive( 'A' )
llIsReady = oDrive.IsReady
```

We can find out almost anything we need to about a drive by interrogating the properties of the drive object. If we need to know how much free space is available on the drive, we can do that easily by accessing either its *AvailableSpace* or *FreeSpace* properties. We can determine what type of drive we are dealing with by looking at its *DriveType* property. This is a numeric value that can contain one of the following:

- Unknown 0
- Removable 1
- Fixed 2
- Remote 3
- CD Rom 4
- Ram Disk 5

Conclusion

The Windows Script Host can be used quite easily in your Visual FoxPro applications. While it is also true that much of its functionality could be implemented using native VFP code, it would require a lot more code. The examples provided here barely scratch the surface, and there is so much more that the WSH can do for you. To assist your exploration of the Windows Script Host, the Help file, SCRIPT56.CHM, can be downloaded by following the links from the Microsoft Windows Script Technologies Web site (http://msdn.microsoft.com/scripting).

Chapter 11
Deployment

Deployment is a big issue that all developers hopefully face at some point in their careers. Otherwise, there is not much point to doing the development, and in most cases you won't make a living in software craftsmanship. This chapter highlights some tips in polishing an application for deployment, and distributing an application via the native setup tools that ship with Visual FoxPro.

Deployment is the end result of a completed development cycle (requirements, design, develop, and test). The product that is deployed can be a component of a large enterprise-wide application, a quick-and-dirty developer tool, or a tier of an n-tier application. It can be a database conversion, a small enhancement, or bug fix to an existing application, or it can be an all-new application. In reality, it can be anything one person or a team of more than one developer assembles for a customer. It may take 30 minutes based on fixing a bug, or may take a year or more for new system development. It could even be one phase of a many-phase implementation that is scheduled over a period of time.

Deployment is not something that should first be considered after the last of the code is developed and tested. It is a process that needs to be mapped out early in the development life cycle. There are a number of issues that should be addressed to eliminate the number of surprises that affect a successful deployment of an application.

This chapter cannot focus on the hundreds of details that can lead to the ultimate in successful software deployments. We figure it would take an entire book on the subject to do complete justice to the topic. This chapter will address some of the more common deployment questions asked over the years, as well as some tips on how to better deploy applications using the InstallShield Express tool introduced in Visual FoxPro 7, and some tips to ease the use of the Visual FoxPro 6 Setup Wizard.

How do I integrate graphic images into an EXE? (*Example: MF11Main.prg and GraphicSample.scx*)

There are several images that make applications look more polished. The most obvious images are the application icon, toolbars, menus (new in Visual FoxPro 7), splash screen, About window, and wizard images. Other images commonly included in an application are backgrounds for the Visual FoxPro desktop and forms.

The application icon is included in the executable and is displayed as the icon for the main Visual FoxPro frame for applications that are not based on Top-Level forms. The code necessary to change the Visual FoxPro frame icon is:

```
_screen.Icon = "MyIcon"
```

If you do not include an icon in the executable, Visual FoxPro will default to the Windows icon when the application is executed with the Visual FoxPro runtimes. The way to include a custom icon in the executable is to set it up as the icon for the project via the Project

Information dialog (see **Figure 1**) available when a project is open. It is important that the icon selected has both a 16x16 pixel image and a 32x32 pixel image. Both of these images are stored in the same ICO file. The 32x32 pixel image is used by the Windows Explorer when large icons are selected for the file display and also on the file property dialog. The 16x16 pixel image is used for the small icon, list, and details view of the file list.

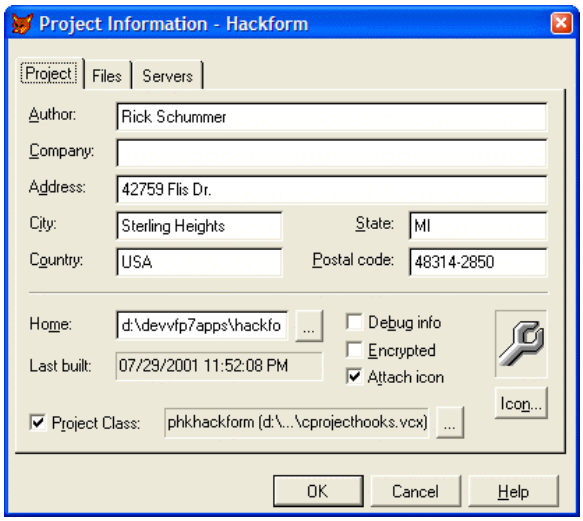


***Figure 1**. The Visual FoxPro Project Information dialog is where you specify the icon compiled into the application.*

All the graphic options discussed in this section are demonstrated in the MF11Main.prg and GraphicsSample.scx included in the chapter downloads available from Hentzenwerke.com.

The form icon is set just like the Visual FoxPro main frame icon, by setting the *Icon* property. This can be set directly in the Property Sheet when editing the form. This is ideal if you have a different icon for each form. Another approach is to make all form icons the same as the application icon. We like to handle this in our base form class in the *Init* method with the following code:

```
this.Icon = _screen.Icon
```

The *Icon* property is stored with a relative path to the icon file unless the icon used is located on another drive from the form. There have been reports that indicate that *Icon* properties with full paths can confuse Visual FoxPro when the icons are included in the EXE. If you are using icons from a different drive in your project and also include the icon in the EXE, it is recommended to strip off the path from the icon property in the *Init()* method.

```
this.Icon = JUSTFNAME(this.Icon)
```

The Visual FoxPro desktop screen and forms can also have images. The **_screen** object has a picture property that will add the image to the desktop. If you have an image that has a pattern that looks good repeated, this is the way to go. If you have an image that you want displayed once like a company logo, then you will want to add an image object to the Visual FoxPro desktop.

```
_screen.AddObject("imgFoxHead", "image")

WITH _screen.imgFoxHead
   .Picture = "FoxBak.gif"
   .Stretch = 1    && Isometric
   .Height  = 300
   .Width   = 420
   .Left    = (_screen.Width/2) - (.Width /2)
   .Top     = (_screen.Height/2) - (.Height /2)
   .Visible = .T.
ENDWITH
```

You can position the image anywhere on the desktop. The code sample centers the image in the middle of the desktop. The image will remain in a static position unless you programmatically change it, even if you change the size the desktop.

How do I create graphic images?

We have a library of images (icons and pictures) that we use in all our applications. We have either purchased or created them during the development of past projects. These common images do not need to be re-created for each customer or application. We typically leave the creation of the project-specific images for the end of a project since most of the effort of development should be directed toward solving the business problem.

We have used the ImageEdit tool that shipped with Visual FoxPro 5 to create and edit icons because it works, and it performs scaling when pasting icon images into the editor. Another popular icon editor is MicroAngel, available from www.impactsoft.com. We have found that editing the 32x32 pixel image first, and then copying it to the clipboard and pasting it into the 16x16 pixel image works best. There are plenty of commercial and freeware icon editors available; just be sure to get one that minimally allows you to edit both of these images.

Each release of Visual FoxPro comes with a set of icons as part of the product. Edit any icon to see how they are assembled. You can even edit one and save it to alter the look to your needs. If the license of an icon package that you purchase allows this, it can be a great way to save time.

The easiest way to create graphics might be to have a professional do it for you. This is what graphic artists do and can add a professional look to your applications. If a graphic artist needs to be contracted, the sooner you can get them involved the better.

There are many sources of images for you to purchase. We use JPEGs (.jpg) and GIFs. We like the JPEGs and GIFs over bitmaps for two reasons. The first is the size of the images; JPEGs and GIFs are compressed, while bitmaps are substantially larger. The other reason is that the JPEGs and GIFs are Web-ready so the images are reusable for Web sites or a Web interface to the application data. The key to purchasing images is that you have the license or right to distribute them. The new Microsoft Image Editor (it comes with several

Microsoft packages including Visual Studio) works satisfactorily for creating and editing JPEGs and GIFs.

How do I deploy graphic images?

There are two methods to deploy images with Visual FoxPro applications. The first is to include the image in the executable; the second is to make sure the images are excluded. You can have the images in the project file with either approach; you just marked them excluded with the second approach. So what are the advantages and disadvantages of each technique?

The advantage of including the images in the executable is that the images files do not need to be distributed separately. This reduces the number of files you have to keep track of when producing the setup. The other advantage is that you do not need to worry about pathing issues to the image directory since Visual FoxPro will find them in the EXE. The big disadvantage when including the images is that it will bloat the size of the executable. Each byte of image adds a byte to the executable size. If you have a megabyte of images included in the project file, you will have an extra megabyte added to the EXE that is shipped to the customer. If the customer downloads the executable it will take longer to download. If the installation is on a LAN so it is accessible to all workstations it will be an extra megabyte that is pulled over the wire each time the executable is started. It is also an extra megabyte of memory that the executable will need when running on the workstation. The same goes for COM objects on the workstation or Web server.

Excluding the images from the executable will produce smaller executables, but requires the developer to track which files need to be distributed and make sure that the executable will have the images on the path.

We take a mixture of the two techniques in our applications. Icon images are typically small and rarely add much to the size of the executable. We try to keep the graphics to a minimum and make sure we compress them as much as possible. We like to exclude images like a company logo for vertical market apps (so each company purchasing our apps can have its own logo). Finding a balance is important and is handled for each specific application we develop.

How do I get the version details from the executable?

(*Example: CH11.vcx::cusGetFileVersion and FileVersion.scx*)

The release of Visual FoxPro 5.0 introduced internal version information in Visual FoxPro executables (EXE/DLL). The version information includes application version number, and text for comments, the company name, a file description, legal copyrights and trademarks, a product name, and language id. This information is entered through the EXE Version dialog (see **Figure 2**) or through the Project Object Version properties. We recommend the minimum properties to include in the executable to be the version number, company, copyright, and product name.

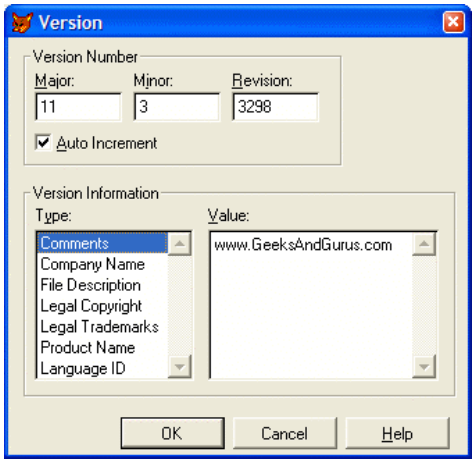


***Figure 2**. The Visual FoxPro EXE Version dialog is where you can enter version information.*

This information can be accessed via the **AGETFILEVERSION()** function in Visual FoxPro 6/7. If you are using Visual FoxPro 5.0 you will need to use the *GetFileVersion()* function included in FOXTOOLS.FLL. Here is a code example on how you can generically get the version information.

```
* cusGetFileVersion.GetAppVersionExecutable()
LOCAL lnCounter, ;
      lcSys16Value, ;
      lcTempAppName

* Process the file name for the APP or EXE
this.lAppFound        = .F.
lnCounter             = 0
this.cAppNameToSearch = UPPER(this.cAppNameToSearch)

DO WHILE(.T.)
  lcSys16Value = SYS(16, lnCounter)

  IF EMPTY(lcSys16Value)
    lcTempAppName = SYS(16,0)
    this.cAppName = SUBSTR(lcTempAppName,RAT(" ", lcTempAppName)+1)
    EXIT
  ELSE
    lcTempAppName = lcSys16Value
    this.cAppName = SUBSTR(lcTempAppName,RAT(" ", lcTempAppName)+1)

    DO CASE
      CASE this.cAppNameToSearch+".EXE" $ UPPER(lcSys16Value)
        this.lAppFound    = .T.
        this.cRunType = "EXE"
        this.cAppName = lcSys16Value
        EXIT
      CASE this.cAppNameToSearch+".DLL" $ UPPER(lcSys16Value)
        this.lAppFound    = .T.
        this.cRunType = "DLL"
```

```
        this.cAppName = lcSys16Value
        EXIT
      CASE this.cAppNameToSearch+".APP" $ UPPER(lcSys16Value)
        this.lAppFound    = .F.
        this.cRunType = "APP"
        this.cAppName = lcSys16Value
        EXIT
      CASE this.cAppNameToSearch+".VCT" $ UPPER(lcSys16Value)
        this.lAppFound    = .F.
        this.cRunType = "VCX"
        this.cAppName = this.cAppNameToSearch + ".VCT"
        EXIT
      CASE this.cAppNameToSearch+".SCT" $ UPPER(lcSys16Value)
        this.lAppFound    = .F.
        this.cRunType = "SCX"
        this.cAppName = this.cAppNameToSearch + ".SCT"
        EXIT
    ENDCASE
  ENDIF

  lnCounter = lnCounter + 1
ENDDO

* RAS 07-Jul-1998 Modified to use new built in functions for GetFileVersion,
* Removed all the FoxTools code
IF This.lAppFound
  lnRetVal   = AGETFILEVERSION(this.aGetFileDetails, this.cAppName)
ELSE
  DIMENSION this.aGetFileDetails[15]

  this.aGetFileDetails = ""
ENDIF
```

There are a number of ways that the cusGetFileVersion can be implemented. These are documented in the class zzAbout() method and in the FileVersion.scx example.

We always include the version information on the About form. If a customer calls with a problem we can see which version of the app they are running. We use the Comment property of version to plug our company Web site

Where should I install my application ActiveX controls?

ActiveX controls can cause developers problems when it comes to versioning issues. We can all relate to the technical support call from the customer that follows the pattern described in one brief discussion:

"I have a problem with this other application ever since I installed your application. It is causing an error on some TreeView control. I called their technical support and they said that they only support version 6 of this control and your application installed version 7. What are you going to do to fix this problem?"

We dislike taking this kind of support call, and we know that they are inevitable if you are using common ActiveX controls either provided with Visual FoxPro or ones that you purchase from a third-party provider.

To reduce the number of support calls and to follow the Windows logo standard, we have adopted a standard. We have two folders to load our application ActiveX controls and components. These folders are based in the folder that the operating system understands as "Common Files." This can differ on each user's machine based on preferences and native language. Typically it is found in the Program Files folder. We create a shared folder for our company in the Common Files folder.

If the components are specific to an application, we install them in a folder under our company shared folder, under the application name, in a Component folder. Here is an example:

```
C:\Program Files\Common Files\GeeksAndGurusShared\OurCustomApp\Components\
```

If the controls are commonly shared across a suite of apps we developed for the customer, we will install them into a directory patterned after this directory structure:

```
C:\Program Files\Common Files\GeeksAndGurusShared\Components\
```

The current install tools provide you a reference to the Common Files directory, which simplifies the installation. It keeps the System32 folder cleaner and hopefully there will be fewer support calls about any versioning issues. The Visual FoxPro 6 Setup Wizard forces the System32 directory route, so you will need to use a custom program if you want to use the old wizard and the new standard folders. Either way, the Registry handles where to find them so that is a non-issue.

Another thing to consider when building the installation process is to see if you can mark the file to only be installed if it is a newer version. Many, if not all of the latest install building tools provide this feature. This can help with two issues. The first is that it can save a potential reboot of the user's machine since some ActiveX controls require the computer to be restarted after they are installed. The second advantage is that the installation will run faster.

Where do the Visual FoxPro runtimes have to be installed?

Visual FoxPro developers have been trained that the runtimes have to be in the Windows System directory. This is where the Visual FoxPro 6 Setup Wizard installs them. The truth is, they have to be available on the Windows Path, can be in the same folder as the executable, or can be installed anywhere and specified using the –D parameter to the executable.

The runtimes can be installed with the EXE on the network or on the client workstation. The consideration of loading the runtimes on the workstation is significant. Visual FoxPro can definitely access the workstation hard drive much faster than pulling the runtimes from the Local Area Network (LAN) file server or over a Wide Area Network (WAN). We always recommend that the runtimes be installed on the workstation for performance reasons. The issue needs to be addressed anytime a new version of the runtimes is released (via a service pack from Microsoft). If you upgrade the development environment, you will need to upgrade the production environment. This means that the runtimes have to be reloaded on each workstation. This can be quite a chore for a company's support staff.

There is no reason to "install" the runtimes via InstallShield Express, the Setup Wizard, or another installation package. There are no Registry entries to update during the installation. The runtime files can be copied from one machine to another or from a CD or Zip disk to the hard drive.

The drawback of the –D parameter is it requires a shortcut to the executable. If the user sets up a shortcut and forgets the parameter, you could see different results. The big advantage of using the –D parameter is that it allows for multiple runtimes modules to be on the same workstation. This is important to know if you have releases of different apps on different versions of Visual FoxPro (service pack deployment issue).

How do I know which runtime files are being used?

Since we can install multiple versions of the Visual FoxPro runtime files in different directories, we might want to know within an application, which set of runtime files are being used. The directory of the runtimes is retrieved using the `SYS(2004)` command.

How can I distribute new versions of the runtime files?

Periodically Microsoft will update Visual FoxPro with a version release or a service pack. Applications that are updated and released after a Visual FoxPro version upgrade will require all the runtime files to be shipped with your application. The service packs can include updated runtime files that need to be release with the updates to your custom applications.

It is very important to keep the runtimes in sync with the development version of Visual FoxPro. One example of a problem could be delivering an application using edit boxes with the original Visual FoxPro 7 runtimes. Microsoft released a hot fix for the runtimes soon after the release of Visual FoxPro 7. If you do not update the runtimes at customer sites, they will not have scrollbars in the edit boxes on their forms. This was corrected in the hot fix (and included in Service Pack 1).

So what files do you need to distribute? There are a few directories to check for new runtime files. Your directories could be different depending on the operating system and the directory structure you installed Visual FoxPro.

- C:\Program Files\Common Files\Microsoft Shared\VFP\ contains the VFP7R.DLL, VFP7T.DLL, VFP7RCHS.DLL, VFP7RCHT.DLL, VFP7RCSY.DLL, VFP7RDEU.DLL, VFP7RENU.DLL, VFP7RESN.DLL, VFP7RFRA.DLL, VFP7RKOR.DLL, VFP7RRUS.DLL, VFP7RUN.EXE, FOXHHELP7.EXE, and FOXHHELPPS7.DLL.
- C:\Program Files\Common Files\System\Ole DB\ contains the VFPOLEDB.DLL.
- C:\Program Files\Common Files\Microsoft Shared\Merge Modules\ contains the merge modules used by InstallShield Express and other install tools that leverage the Windows Installer technology. These files include VFP7RCHS.MSM, VFP7RCHT.MSM, VFP7RCSY.MSM, VFP7RDEU.MSM, VFP7RESN.MSM, VFP7RFRA.MSM, VFP7RKOR.MSM, VFP7RRUS.MSM, VFP7RUNTIME.MSM, VFPACTIVEDOC.MSM, VFPHTMLHELP.MSM, VFPODBC.MSM, and VFPOLEDB.MSM. The merge modules are not directly distributed to the customers, but are used by install tools like InstallShield Express and Wise for Windows Installer.

You can review the list each time a fix is delivered by Microsoft. Now that we know which files can change, the question begs, how can you redistribute the runtime updates to the client sites? There are a couple of options.

The obvious way is to rebuild the distribution files via your installation tool of choice. If you are using InstallShield Express – Visual FoxPro Limited Edition, the new runtime files will be available in the merge modules. Include the correct merge modules, rebuild the setup, test, and distribute. This is the safest and possibly the most polished way to redistribute the runtimes.

There is nothing limiting you from directly copying the updated files to the workstation. You can copy the changed runtime files from a network server to each workstation via something as simple as a DOS batch file, create a self-extracting Zip file to be run on each workstation, post them on a Web page with instructions to download them, burn them on a CD with an auto play that copies them, or have a process check for new updates each time the application is started to see if an update is available. The runtime files only need to be registered using REGSVR32.EXE if your application uses Active Documents. Taking this approach might be the easiest way if you are onsite at a client's and just need to move a couple of runtime files to a couple of workstations.

The method of getting the runtime files to the client workstations will depend on many factors. You will need to evaluate the problem and determine the best mechanism for the situation. You have many alternatives. In the past many developers thought that they needed to build a complete install package each time new runtimes needed to be loaded.

How do I run a different Visual FoxPro runtime language resource?

Visual FoxPro 7 ships with nine runtime language resource files (DLL extension)—they're listed in **Table 1**. These are available to run both the development and runtime versions of Visual FoxPro in a language that is different from the default.

***Table 1**. Language resource files shipped with Visual FoxPro 7.*

Language	Runtime file
English	VFP7RENU.DLL
German	VFP7RDEU.DLL
French	VFP7RFRA.DLL
Spanish	VFP7RESN.DLL
Simplified Chinese	VFP7RCHS.DLL
Traditional Chinese	VFP7RCHT.DLL
Russian	VFP7RRUS.DLL
Korean	VFP7RKOR.DLL
Czech	VFP7RCSY.DLL

Selecting a different language is available via the –L parameter to the VFP7.EXE or your own custom EXE. Make sure that you include a full path if the file is not available in the startup directory or on the search path. There are no spaces between the –L parameter and the file name.

```
C:\MegaFox\Chapter11\CH11.exe -Lc:\Progra~1\common~1\Micros~1\VFP\VFP7rdeu.DLL
```

The example in **Figure 3** demonstrates the native Visual FoxPro menu now displayed in German.

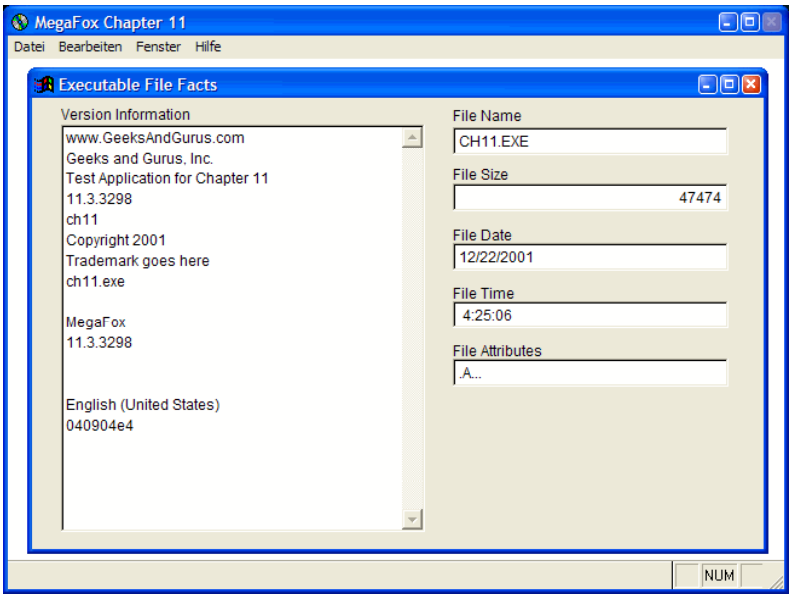


***Figure 3**. The menu has changed for the application when using the German language resource runtime file.*

The language resource files will not translate your custom captions; it only changes the native Visual FoxPro dialogs and menus. You will still need to perform some form of translation for your own labels and captions.

What executable format can I release my application?

Visual FoxPro has four different executable file formats that can be released: EXE, APP, DLL, and FXP. Each of the formats can be used in the various types of implementations (desktop, client/server, n-tier, COM, and Web).

APP files need either the Visual FoxPro development environment or can be called from another runtime Visual FoxPro EXE. If your clients have the development environment loaded on a computer, you can run the APP by passing the APP file name as a parameter to the Visual FoxPro 7 executable. Here is a sample program call:

```
VFP7.EXE MyCustom.app
```

Visual FoxPro APP files are also used to implement Active Documents. This requires the main Visual FoxPro runtime file be loaded on the client workstation and registered. This is the only time that the Visual FoxPro runtimes need to be registered. The VFP7R.DLL is self-registered with REGSRV32.EXE. Active Documents can be executed via the VFP7RUN.EXE as

well as the VFP7R.DLL runtime files. One advantage of shipping an APP file is that you can compile a component and just deliver the component. This is a common approach for a suite of applications. You deliver one main executable and a set of different APP modules that provide the various features. If only one of the features is changed, you send the one APP instead of the entire executable.

The Window executable (EXE) is the most common Visual FoxPro executable implementation. It is an APP file with a Windows boot segment added to the APP file. This file can be executed from a shortcut, from Explorer, and even from the Windows DOS command prompt. It requires all the runtime files (VFP7R.DLL, and VFP7R<LANGUAGE>.DLL, which is the corresponding language resource file) to be available. Visual FoxPro EXEs can be called from other Visual FoxPro executables (using **DO <EXE>** and **RUN**). Objects in the EXE can also be instantiated by Visual FoxPro and non-Visual FoxPro programs (via the **SET CLASSLIB TO <class> IN <EXE>** and **CREATEOBJECT()**). Visual FoxPro EXEs can also be executed within the development environment in the same manner as the APP files. If there are classes compiled in the EXE marked OLEPublic, then other Visual FoxPro and other COM clients can instantiate the Visual FoxPro classes and manipulate the class properties and execute methods. The advantage of shipping an EXE is that you can literally ship one large file to your client installations. There is no need to track a bunch of source files to send. The disadvantage is that it takes longer to ship the entire EXE over the Web or longer for it to be downloaded by the customer.

The Visual FoxPro DLL is an in-process COM object. The decision you will need to make for implementation is whether you will be using the standard single-threaded runtime or the multi-threaded runtimes. The single-threaded runtime simply blocks more than one object from executing code in the DLL. It queues up the requests and processes them in sequence. When the first object completes the property assignment or method execution, it processes the request from the second process. If the object method takes a half a second and 1,000 objects simultaneously make a request, then it will take 500 seconds to process all the requests. The multi-threaded runtimes (VFP7RT.DLL) will not block the other processes from running. It will time slice the requests. The multi-threaded runtimes are also lighter and have no capability to display a user interface. Therefore, many capabilities to output messages and data to the screen have been eliminated and the DLL is smaller in size. The multi-threaded runtime library will also take advantage of multiple processors in the computer.

The compiled Visual FoxPro programs files (FXP) can also be released. These programs need to be run in the Visual FoxPro development environment, called from another executable (EXE or APP), or can be run directly from the VFP7RUN.EXE. The FXP can call all other source code objects like forms, reports, visual classes, and so forth. The advantage of shipping individual compiled programs is that you can implement a component or feature quickly, without the need to kick all the users out of the application. The disadvantage is that you need to distribute many files instead of one EXE or a few components. You will also be sending source code when delivering forms called from FXPs.

What installation scheme should I use?

After the directories are created and the correct files are placed in them, you need to determine what installation scheme you need for the release. We are sure there are numerous schemes to create an installation process. The following ideas are ones we have found successful for our

Visual FoxPro application deployments. We may combine several installs into one package, or we might ship separate installs based on the customer's environment or needs. We have separated the upcoming discussion based on the different options that we can include in an installation package.

File Server Install

This is the main application installation and is included in nearly every installation package that is delivered to the customers. It includes all the core application executable files needed to run the application. This is the "base" installation and includes all the files found in the installation root, system, sound, images, and any other directories needed by the application.

This install will typically be used in a network environment. The installation loads the core application files on the file server in a specified directory. Once the files are loaded the users will have access to the application provided that they have network access and rights to these files. Using this scheme also requires that all the workstations have the Workstation Install (discussed next) loaded so they have the Visual FoxPro runtimes.

This installation will also work on a client PC for a single-user application provided that the files from the Workstation Install and the Data Install are also installed. Many developers who have single-PC clients use this technique all the time.

Workstation Install

If the application is loaded on the file server, is it ready to be executed by the connected workstations that have security access? Not necessarily. Each workstation needs additional files loaded. These include the Visual FoxPro runtimes, any ActiveX controls, and the Help system engine. You could go around to each workstation and reload the File Server Install (discussed in the previous section), but this can be time-consuming and unnecessary. The idea of the Workstation Install scheme is to load only the files needed. We have broken the Workstation Install into five installs, all included in separate directories on one CD-ROM. You may find there to be more or less than this depending on your needs.

The first install is the Visual FoxPro runtimes. We find that application startup performance is best when the Visual FoxPro runtimes are loaded on each client workstation. Do the runtimes really have to be on the client workstation? No, see the –D directive to the VFP7.EXE (and consequently, your custom executable), but it is faster since you are not pulling 4MB down the pipe every time you start the app. This install will load all the Visual FoxPro runtimes including VFP7R.DLL, and VFP7RXXX.DLL (where the xxx is the language of the Visual FoxPro version you have—for instance, enu for English).

The second install is the multi-threaded runtimes. This loads the new VFP7T.DLL file for multi-threaded COM objects. Not all applications require the multi-threaded functionality, so installing this for your customers may not be required.

The third install is for the HTML Help Engine runtimes. These files are only needed if you include a CHM file with your application. If you decide to build WinHelp files (HLP) or a table-based Help file, you will not need this installation.

The fourth install is for the Visual FoxPro ODBC driver (VFPODBC.DLL and VFPODBC.TXT file) or OLE DB driver (VFPOLEDB.DLL) and others if needed. This gives your users a way to analyze their data via tools like a spreadsheet, perform mail merges from a word processor, or build their own queries via an end user database or tool.

The fifth install is the ActiveX controls. The key to this install is to make sure the ActiveX controls included in the application are installed on the workstation. These files are loaded into the appropriate directory and installed in the Windows Registry. You will need the ActiveX controls loaded on the computer that the installation is being built on. It is important to note that the ActiveX controls loaded during an install can be the ones included with Visual FoxPro and Visual Studio as well as any third-party controls purchased.

All of these installs are copied to one CD-ROM in separate directories. You need to do this because the install tool can name the setup (SETUP.EXE, SETUP.INF, SETUP.INI, SETUP.LST, SETUP.STF, SETUP.TDF) and corresponding cab (SETUP1.CAB, SETUP2.CAB) files identically for each install. You may decide to customize this CD as well for a specific customer. It may be that you build one app with various ActiveX controls and another app for a different customer without ActiveX controls, or a different app with different controls. This CD (or copies of it) can be passed around from computer to computer. We also recommend the CD be dated, and note the Visual FoxPro version and Visual FoxPro service pack that the runtimes apply. It sure can be embarrassing to have that new Session class given to us in Visual FoxPro 6 Service Pack 3 not be available with a new executable running on prior versions of the runtimes.

Data Install

Obviously this installation section is for the application data. The questions that need to be asked though may complicate this seemingly easy setup. What files need to be sent? Is this Visual FoxPro local data or are we using a SQL back-end database? What tables need to be pre-populated? What files can be generated at the customer site? What about installations that already have previous installations with data loaded?

The initial installation will require that the all the application data be installed and this data can be found in the installation data directory. We like to keep this installation separate, especially for a new service pack release. Vertical market applications will like this scheme as well since it allows a development shop to build a single installation package for new customers and existing customers getting a new version.

Web Server Install

A Web Server Install may mirror a File Server Install scheme in many ways. There are, however, many differences that your application may encounter. You will likely be installing the multi-threaded runtimes for scalability, COM components or an EXE, HTML files, and be making Web Server settings (via executable or another manual settings) like scripting files, user security, and the mapping of drives to the data.

How do I package the install?

Now that we have developed schemes for the installation, we need to determine how we will package it. We are not referring to the box that the CD is delivered in. The marketing experts best handle this. We are suggesting that you need to think out what installs discussed in the previous section need to be packaged up and sent to the customer.

Typical brand-new installs for a LAN-based application require the File Server, Workstation, and Data Install schemes. Updates may only require the File Server Install. On the other hand, if the executable is built with a new version of Visual FoxPro, you need to ship at least part of the Workstation Install. A Web Server may only need some new HTML files,

so you can skip the need to update COM objects. Single-computer customers may require that the File Server Install and the Workstation Install be combined into one installation process.

The packaging of the install is as important as developing perfect software since it is likely to be the first impression that most users have of the application in production. We are not talking about the people you developed the software specs and performed acceptance testing with; we are talking about the possible hundreds of end users who actually get the package loaded on their computers.

What are some handy utilities to ship?

While the main goal of the developer is to install the files needed for the customer's application, there are a number of developer utilities that can assist in the installation and ongoing maintenance that is likely to occur. The following are concepts for the tools that we have developed for the types of applications we deliver to our customers.

Reindex and Database Updater

Initial releases usually start with an empty database or have a data converter preload the database. What happens when an upgrade release is made and the latest alterations to the database tables, indexes, views, and relations need to be implemented? You could write a custom program each and every time that makes these changes via the powerful `ALTER TABLE` and `INDEX ON` commands. You can also keep track of each change made to the data and make sure that this custom program is updated every time a change is made in development. Even for single-developer projects this can be a tough task, and the odds of missing one critical change is nearly even. Multi-developer projects get to be more than a challenge in this respect; they become a communication nightmare.

Keeping track of these changes can be automated. We do not want this to become a commercial for the Stonefield Database Toolkit (SDT, available at www.stonefield.com), but we find so much value in this product that we have to give it a plug. It keeps track of your changes to the data structures in the DataBase Container eXtensions (DBCX) and SDT metadata. SDT provides a *NeedUpdate()* method to check for differences in the metadata and what the structures are in the database. If there are differences you can run the SDT *Update()* method and the structural changes are applied to the database tables. This means that new columns can be added or removed, column name changes are applied, and code pages can be changed. The same is true for indexes. It will also create new tables if they do not exist. This is all handled based on using the DBCX extensions to the database via the Stonefield Database Toolkit or your own developed tool (yes, you can develop one since DBCX is a published standard). The key to this is to validate the database extensions before you ship out the metadata with the release (a lesson learned on our very first release with this product). SDT can be used in initial installations to completely generate all the tables as well.

There is another option to solve the database changes, and that is to simply make the changes manually. If you develop onsite with the application you can just use Visual FoxPro live on the data and make the changes. We hear of this all the time. We are just not the kind of developers who trust ourselves to remember to make the changes in the same fashion as we did in development. There are surely plenty of war stories to be told that would convince you not to do this. On the other hand, emergency fixes that can be done to keep a customer alive are made all the time.

One thing that SDT does not handle that you will need to consider for all types of releases is data conversion. Even if you have an automated way of updating database structures and the like, you will need to consider a mechanism of populating new fields, converting data from old tables, and cleaning up data that might violate new field or row level rules. You might need a separate program that cleans up the data before implementing a new field or row level rule for a table.

GenDBC/GenDBCX

If you are not using SDT and/or want a mechanism to generate the database and all the table structures, views, indexes, and relations, take a look at GenDBC (included with each release of VFP) or GenDBCx. GenDBCx is a third-party tool written by Steve Arnott, which is available for free. It can be downloaded from www.dfpug.de/forum/incat.htm?nsec=8 or www.webconnectiontraining.com/tools.htm. Both of these tools generate Visual FoxPro program code that will build the database from scratch. Just like the SDT Update process, neither of these tools populates the tables so you will need a mechanism to accomplish this task.

Checking next id table

Developers who use surrogate keys (meaningless integers or characters that uniquely identify a record in a table) will have a table that contains the next key for tables. Periodically these tables will get misaligned with the real data in the tables. This can happen because the developer writes bugs in their applications; tables get zapped moving from development to production without updating the next surrogate key table, incorrect referential integrity rules, or the planets being out of alignment.

For whatever reason, the next id table needs to be synchronized with the data in the tables. This process will need exclusive use of the database and each table. The general algorithm is to get the maximum key value from the table via code like:

```
SELECT MAX(nTablePK) ;
   FROM Customer ;
   INTO ARRAY laMaxID
```

Once you have the maximum id for the surrogate key, the next id table record for the table is updated with this new value. It is a good idea to run this process for all the tables in the application periodically. One red flag that indicates that it might be time to give this process a run is the constant calls from a customer that they cannot add any records into any form because they are getting a message indicating duplicate keys values.

A way to avoid having a program like this is to use Globally Unique IDentifier (GUID) keys. The GUID is a 16 alphanumeric string. It takes up more space and creates larger keys and is slower than integer keys, but they are unique, even across different locations, which can come in handy if you need to consolidate data from the same table that is located at different sites.

Configuration/control table updater

Many applications have an INI file or a configuration table. When new options are added a mechanism to get these options into an existing application needs to be considered.

Personally we prefer to work with tables since we work with these all day and Visual FoxPro provides plenty of commands to manipulate data. Adding new options is as simple as an **APPEND FROM** or running code to do **INSERT INTO**. Updates are as simple as a **LOCATE** or **SEEK** and using **REPLACE**s. We can also write code that does a SQL Update. The INI text files are also easy to work with since Visual FoxPro has low-level file input and output commands to manipulate text files. There are also Windows API calls to INI files available for developers with this knowledge.

The important item to note is that you develop some mechanism to update this information so the customer application does not malfunction when new options or features are added.

InstallShield Express for Visual FoxPro tips (*Example: CH11.ism*)

The full name of the replacement for the Setup Wizard used in previous versions of Visual FoxPro is: Install Shield Express – Visual FoxPro Limited Edition. When we hear "limited edition" it is usually used in the context of something special that might be collectible or cherished. In the case of InstallShield Express, Limited Edition means that it has a limited feature set. It is the "lite" version of InstallShield Express that is available from InstallShield Software Corporation.

InstallShield Express is a tool that builds installation routines that run on the Windows Installer technology included with Windows. It uses a more modern interface than the old 16-bit installer that the Setup Wizard generated.

Where do I find InstallShield Express?

InstallShield Express – Visual FoxPro Limited Edition is included on the Visual FoxPro CD-ROM. It is not automatically installed when you select all the files when installing the VFP. It is a separate installation in the same CD. You need to select the Install InstallShield Express option from the Visual FoxPro install startup screen (see **Figure 4**).

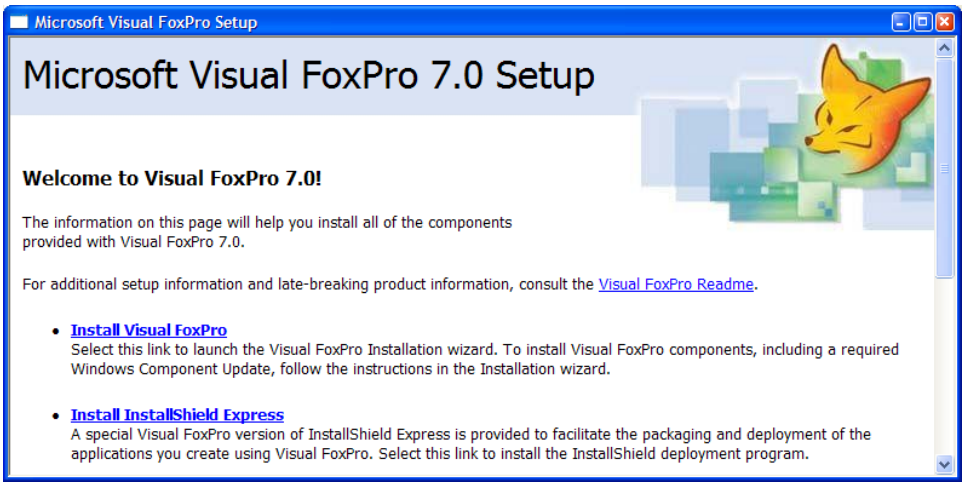


***Figure 4**. InstallShield Express – Visual FoxPro Limited Edition installation is started from the Visual FoxPro 7 installation startup screen.*

What are the advantages of using InstallShield Express over the Setup Wizard?

InstallShield Express (ISE) is a big step forward in flexibility and is a full 32-bit application. It provides a number of features long desired by developers who have used the tried and true Setup Wizard.

The first advantage is that you no longer need to copy all of the files that are distributed in the build into a separate directory. ISE lets you specify which files are to be distributed, and where on the target system they go.

The Setup Wizard allows only for all files or no files. You did not have a choice in the matter. InstallShield Express allows the users to pick what "features" they want installed, much like the typical options: typical, all, or custom. Picking custom will allow the user to further tailor what is installed. You can define what these options are and what files will be installed when the option is specified.

InstallShield Express provides generic references to all of the Windows folders. The Setup Wizard allowed developers to install files to the application folder and the Windows System folder (primarily for FLLs, DLLs, OCXs, and the Visual FoxPro runtimes). Now you have references like INSTALLDIR, DATABASEDIR, and ProgramFilesFolder to direct files to predefined folders based on the folder structure used on the user's machine. See the section on "How do I leverage the default Windows directories?" in this chapter for more details.

A dynamic setup mechanism allows the developers to select the screens used in the setup. This allows you to display pages for a license agreement, a ReadMe text file, the entry of the user name and company, where the installation files are located, where the data is located, provide for a custom setup, and determine what is on the setup complete dialog. The Setup Wizard only allowed you to customize the name of the application and copyright information.

InstallShield will not only install selected ODBC drivers, but will install pre-built datasources (DSNs) as well. With the Setup Wizard you needed to write custom code with calls to the Windows API and have this executed as a post-install routine.

InstallShield knows where the Windows font folder is and can install fonts that you need to install with your application. It has built-in capability to store things to the Windows Registry, modify INI files, and set up file extension associations. Again, with the Setup Wizard you needed to write custom code with calls to the Windows API and have this executed as a post-install routine.

What are the disadvantages of using InstallShield Express vs. Setup Wizard?

At first glance the InstallShield Express product looks incredible and a giant leap forward. There are a couple of show-stoppers that make one think it is not a complete replacement product for the old-fashioned Setup Wizard included with Visual FoxPro 3, 5, and 6.

There is a feature called Upgrade Path. This feature is only available in the full edition of InstallShield Express, not the in limited edition. The Upgrade Path feature is a way to configure how the second installation of your product is going to run. Will it replace files, update only newer files, and determine which versions it will upgrade? When you run a different build of the custom InstallShield, it prompts you with the message shown in **Figure 5**.

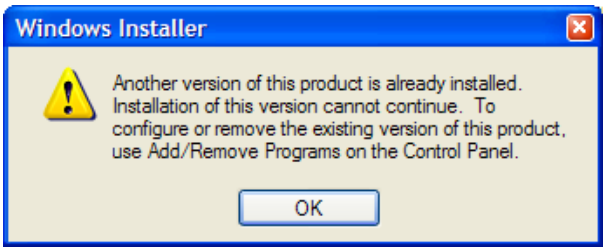


***Figure 5**. InstallShield Express – Visual FoxPro Limited Edition installations require you to remove the previous installation before reinstalling.*

This means that the users will uninstall the executables, the shortcuts, data, remove Registry entries, and any other item that was installed with the previous install. The Setup Wizard allowed for reinstallation (complete overwrite) or removal of the previous installation. This one "feature" makes the entire product absolutely useless unless you only intend on the custom product shipping once or it has no negative effect on uninstalling all the files before reinstalling the upgrade. This is different from reinstalling the current version. InstallShield does provide a mechanism to modify and repair existing installs for the same version.

The other feature that is missing (actually it is one of the "limitations" of the Limited Edition) is that it cannot run a post-installation process. While many reasons we ran post-installation executables have been integrated in the base tool (Registry updates, creation of shortcuts), we still need processes run to make database structural changes or data conversions. The users can manually run these routines, but it is something they will need to remember using an InstallShield Express routine.

How do I upgrade to the full version of InstallShield Express?

The quick answer is that you cannot "upgrade" the InstallShield Express – Visual FoxPro Limited Edition to the full version of InstallShield Express. You will need to purchase a full license. InstallShield Express was provided courtesy of InstallShield via Microsoft as a basic method to installing your custom applications, not as a full upgradeable license. For a short time in March 2002, InstallShield did offer a $100 discount to Visual FoxPro developers to purchase the full version of InstallShield Express based on feedback from the FoxPro Community.

How do I leverage the default Windows directories?

A big advantage to using InstallShield Express (and other commercial install packages) is the ability to place files in different directories. InstallShield also provides a list of Windows folders through generic references. InstallShield converts the references into folders by reading the operating system during the actual installation (see **Table 2**). This eliminates any hard-coding of paths.

***Table 2**. Optional Windows folders available in InstallShield Express.*

Folder variable	Description
AdminToolsFolder	Points to the folder where operating system administrative tools are located, obtained from the operating system.
AppDataFolder	Full path to the current user's Application Data folder, obtained from the operating system.
CommonAppDataFolder	Full path to the folder containing application data for all users. An example in Windows XP is C:\Documents and Settings\All Users\Application Data, obtained from the operating system.
CommonFilesFolder	Full path to the Common Files folder for the current user, obtained from the operating system.
DATABASEDIR	Destination for your setup's database files. You can set the initial value for DATABASEDIR and have the end users modify this value during the installation in the Database Folder dialog.
DesktopFolder	Full path to the Desktop folder for the current user unless the setup is being run under NT/2000/XP for All Users, and the ALLUSERS property is set, then the DesktopFolder property should hold the full path to the All Users Desktop folder, obtained from the operating system.
FavoritesFolder	Full path to the Favorites folder for the current user, obtained from the operating system.
FontsFolder	Full path to the Fonts folder, obtained from the operating system.
INSTALLDIR	Destination folder for your setup. You can set an initial value for INSTALLDIR and have the end users modify this value during the installation in the Destination Folder dialog. This property can be set using any of the other system folders.
LocalAppDataFolder	Locally stored application data, obtained from the operating system.
MyPicturesFolder	Full path to MyPicturesFolder, obtained from the operating system.
NetHoodFolderProperty	Full path to the current user's Network Neighborhood folder, obtained from the operating system.
PersonalFolder	Full path to the current user's Personal folder, obtained from the operating system.
PrintHoodFolder	Full path to the current user's Printer Neighborhood folder in Windows NT/2000/XP, obtained from the operating system.
ProgramFilesFolder	Full path to the current user's Program Files folder, obtained from the operating system.
ProgramMenuFolder	Full path to the Program menu for the current user. If the setup is being run under NT/2000/XP for All Users, and the ALLUSERS property is set, then the ProgramMenuFolder property should hold the full path to the All Users Program menu, obtained from the operating system.
RecentFolder	Full path to the current user's Recent folder, obtained from the operating system.
SendToFolder	Full path to the current user's SendTo folder, obtained from the operating system.
StartMenuFolder	Full path the Start menu folder for the current user. If the setup is being run under NT/2000/XP for All Users, and the ALLUSERS property is set, then the StartMenuFolder property should hold the fully qualified path to the All Users program menu, obtained from the operating system.
StartupFolder	Full path to the Startup folder for the current user. If the setup is being run under NT/2000/XP for All Users, and the ALLUSERS property is set, then the StartupFolder property should hold the full path to the All Users program menu, obtained from the operating system.
System16Folder	Full path to the folder containing the system's 16-bit DLLs, obtained from the operating system.
SystemFolder	Full path to the Windows system folder, obtained from the operating system.
TempFolder	Full path to the Temp folder, obtained from the operating system.
TemplateFolder	Full path to the current user's Templates folder, obtained from the operating system.
WindowsFolder	Full path to the Windows folder, obtained from the operating system.
WindowsVolume	Volume of the Windows folder. It is set to the drive where Windows is installed, obtained from the operating system.

The advantages of dynamic paths are obvious. The implementation of the folder variable is handled in InstallShield by entering the value for the property. You can concatenate more than one folder variable if necessary (see the DATABASEDIR properties in **Figure 6**). Just be careful because many of the folders are fully pathed when expanded during installation.

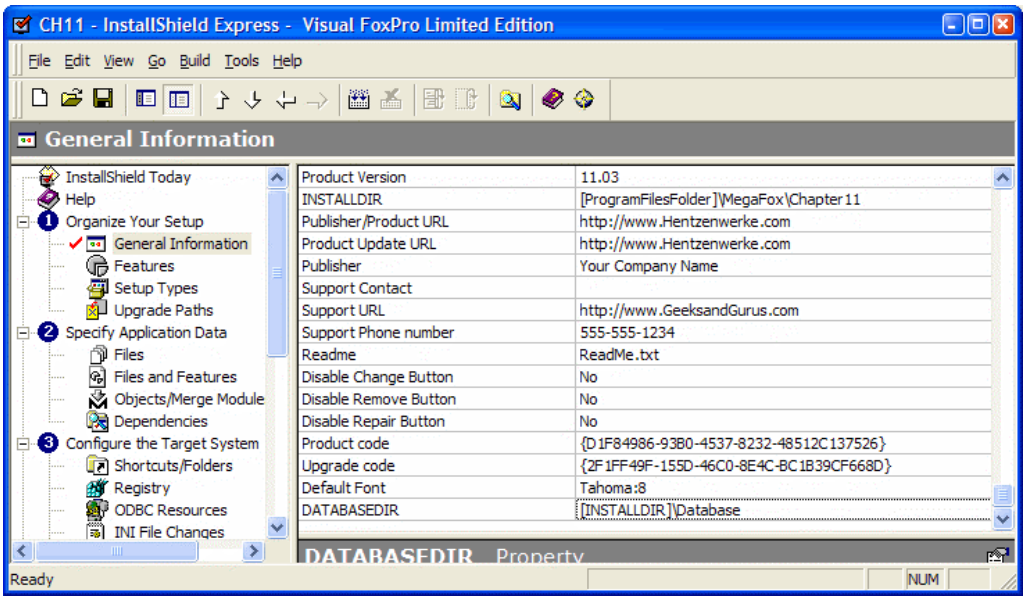


***Figure 6**. You can leverage multiple Windows folder properties assigning any directory property.*

In the General Information section you can specify the developer defined INSTALLDIR and DATABASEDIR (defaults for the installation directories, changeable by the installers). The Files section allows you to add any of the Windows folders by right-clicking on the Destination Computer, and selecting the Show Destination Folder. More than one folder can be added by repeating the selection. The Shortcut/Folders section allows you to add shortcuts to the Start Menu, the Programs Menu, the startup folder, the desktop, or a custom menu. The shortcuts have Target and Working Directory properties. These properties accept any of the folder variables. Registry values and the INI file Target property can utilize the folder variables as well.

How do I work with setup types and features?

Features are file bundles within the install package from the user's perspective. Setup types are predefined categories that are assigned features. The user will select a setup type and behind the scenes the files associated to the setup type through the defined features are installed.

The setup types default to Typical, Minimum, and Custom. If the user selects the Custom option they will be able to pick and choose features that you have included. You can change the captions of the setup types by right-clicking on the setup type to bring up the context menu with the rename option. You will not be able to add new setup types. The first one in the list

will be the initially selected option, so move the options around if you prefer a different default or different order.

The features represent a function, capability, or component of your application and have much flexibility. You can add new features and subfeatures. The assignment of files to the features is made in the Files and the Files and Features sections. The Always Install feature cannot be removed and is included in every install project. It does not show up on the Custom install either. It is designed to always run, regardless of which setup type is selected by the user. Here are some sample ideas for ways you can configure features:

- Application (includes the metadata), Data, Runtimes
- Application, Data, Help, Reports
- Executable, Source Code

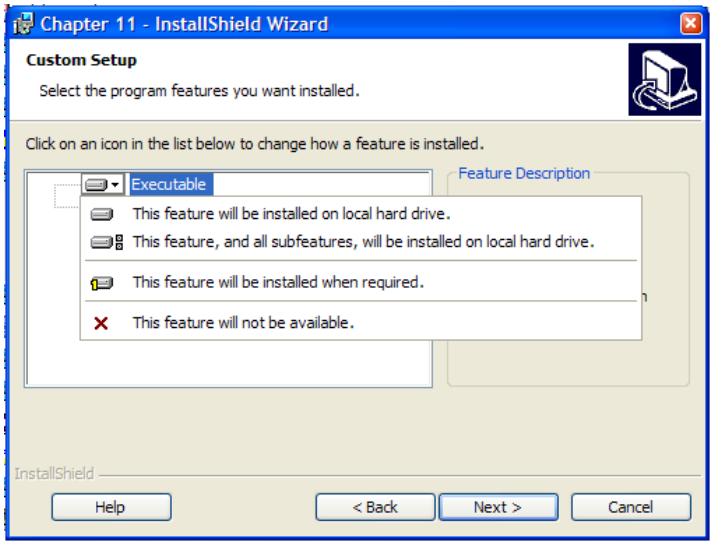


***Figure 7**. The user will be able to select which features are installed if they pick the Custom setup type.*

If the Custom setup is opted by the user, they can customize which features are installed and how they are installed (see **Figure 7**). The user can determine if they want a specific feature installed, not installed, or opt to have it installed at a later time. There is no way to customize the options on the dropdown.

What is a merge module and which do I use for Visual FoxPro installs?

A merge module (MSM file) contains all the files needed to install an application, component, runtime files, or other functionality. All the necessary logic and Registry entries are also included to direct the installer routine. These merge modules save you the time of selecting all

the individual files included in the merge modules and figuring out the dependencies of other files that they require.

The merge modules are supplied with InstallShield Express. The InstallShield Express product cannot create or alter merge modules. You need the InstallShield Developer edition to create or update the merge modules. Updates to the merge modules will be provided by InstallShield or the other software manufacturers. For instance, Visual FoxPro 7 Service Pack 1 shipped new runtime merge modules.

InstallShield Express comes with a number of merge modules for Visual FoxPro developers to include in their custom installations. The merge modules can be selected in the Objects/Merge Modules section. All you have to do is click on the checkboxes for each of the modules you would like included in your custom install. This allows you full control over how much or how little is included in the installation outside of the specific files you picked in the Files section.

A minimum install should include the Visual FoxPro 7 runtimes libraries and the Microsoft Visual C++ 7.0 Runtime Library (see **Figure 8**). There are a number of merge modules to select from for the Visual FoxPro runtimes. Which ones you select will depend on the languages you support. Minimally you will need to check the Microsoft Visual FoxPro 7 Runtime Libraries (VFP7RUNTIME.MSM) and the Microsoft Visual C++ 7 Runtime Libraries (MSVCR70.MSM).

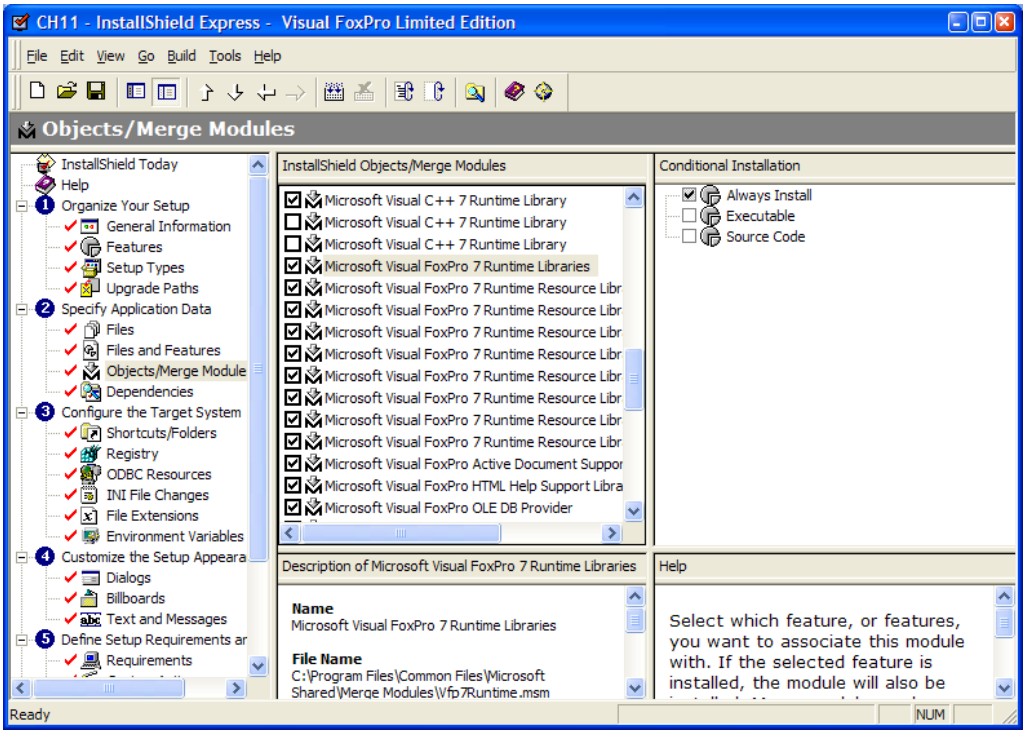


***Figure 8**. Select Visual FoxPro runtimes and VC++ runtimes for a minimum Visual FoxPro custom application installation.*

If you support a language other than English, you also need to select the appropriate resource library (see Table 1 earlier in the chapter for list of resource libraries). The VFP7RUNTIME.MSM file includes the VFP7RENU.DLL, which is used for all English shipping applications. If you want to include support for another localized resource file (VFP7RXXX.DLL), include the merge module containing the localized resource file. For example, include VFP7RDEU.MSM for the German runtime resource file. You will need to look in the merge module description pane (middle, bottom in InstallShield Express) to read the merge module file name.

So why do you need to include the MS VC++7 Runtime Library? To avoid getting a call from a customer who just installed the latest version of your application. When they fire up the application for the first time they would see a message "msvcr70.dll not found." This is a common first time InstallShield Express installation mistake. It is a problem easily missed unless you are testing on a machine that had no previous Visual FoxPro installation. This is one example of why it is nice to have a clean machine to test your installations.

There are three other specific Visual FoxPro merge modules. The Visual FoxPro OLE DB provider makes it possible for both Visual FoxPro and non-Visual FoxPro applications to access Visual FoxPro data using OLE DB or ActiveX Data Objects (ADO). To install the Visual FoxPro OLE DB provider on the customer's machine, include the Microsoft Visual FoxPro OLE DB Provider (VFPOLEDB.MSM) merge module. The older Visual FoxPro ODBC driver is still available for installation via the VFPODBC.MSM merge module. The Microsoft Visual FoxPro HTML Help Support Library (VFPHTMLHELP.MSM) merge module includes both FOXHHELP.EXE and FOXHHELPPS.DLL files needed to support HTML Help within your custom Visual FoxPro applications. In addition to your application-specific CHM file, you might have to include the core HTML Help viewer files.

If your application uses Web Services or the Simple Object Application Protocol (SOAP), you must include the following merge modules:

- SOAP SDK Files (SOAP_CORE.MSM)
- Visual Basic Virtual Machine (MSVBVM60.MSM)
- Microsoft Component Category Manager Library (COMCAT.MSM)
- Microsoft OLE 2.40 (OLEAUT32.MSM)

There are merge modules for the ActiveX controls that ship with Visual FoxPro and previous versions of Visual Studio, as well as the various Microsoft data access technologies. Third-party control and COM providers may also include merge modules with their products for you to include as part of the installation routine. Updated and new merge modules should be available on the InstallShield Web site.

How do I create shortcuts or folders?

InstallShield Express makes it possible for you to create shortcuts and folders both in the Start menu and on the desktop. In addition, shortcuts can be associated with the features that you defined earlier.

1. Navigate to the ShortCuts/Folders section.
2. From the Shortcuts TreeView in the center pane, right-click the node where you want to install a shortcut or folder, and click New Shortcut or New Folder.
3. Type a name for the item you created; this will be the caption for the shortcut.
4. If you created a shortcut, you must specify the Target. In the Shortcut Properties pane (right most), select the Target property, and then select a target folder from the combo box. You can add your own file name to the end of the Target folder selected from the combo box list. The Description property will translate into the ToolTip for the shortcut in Windows 2000/ME/XP. You have the option to associate your shortcut with a feature. Select the Feature property, and then select a feature from the combo box. The Icon File can be an ICO or EXE with the number of the stored icon in the EXE selected being saved in the Icon Index property.

How do I create Registry keys?

If your application uses Registry keys to keep track of user options or application settings, InstallShield Express can add them to the user's machine during the installation. It should be noted that creating Registry keys is an optional step in creating a setup program.

Registry entries are created in Registry hives that categorize Registry entries by function. For example, software options, such as options for Visual FoxPro or your custom application, are contained in the HKEY_CURRENT_USER hive under Software\ <CompanyName>\<ApplicationName> while COM server classes are contained in the HKEY_CLASSES_ROOT.

If the Registry entries exist on the development machine it is as simple as dragging and dropping the Registry entry from the Source Computer's Registry View to the Destination Computer's Registry View. We recommend that you follow the identical hive configuration because it is likely that your custom application will be looking for it in the same place on the destination computer.

If the keys do not exist on your development machine, you can either create them by hand using RegEdit or the InstallShield interface, or programmatically with Visual FoxPro or another tool. These Registry entries will be available in the top pane (source computer). If you want to manually create the Registry entries in InstallShield, here are the steps to follow:

- Right-click the Registry hive on the Destination Computer's Registry View (bottom pane).
- On the Context menu, click New | Key and type a name for the key.
- Right-click the new key.
- On the Context menu, click New, and then select the type of value you want to add to the key.
- Double-click on the new value and enter in the initial value for the entry.

How can I limit the hardware configurations the app will install?

InstallShield provides a number of configuration checks that will stop a user from installing your application if their computer does not conform to the required specifications.

The first check is checking to make sure the operating system (OS) meets requirements. This option allows you to pick all operating systems (not placing restrictions), or pick and choose which operating systems are acceptable. There is one problem with the initial release of InstallShield Express – Visual FoxPro Limited Edition; if you select the operating systems, there is no option for Windows XP. Guess what, it will not allow an install on XP unless you allow all OSes.

The processor option allows you to select all processors, 486, or Pentium or higher. We know that Visual FoxPro will not work on a 486, so we encourage this option to be set at Pentium or higher.

The RAM options allow you to specify the lowest amount of RAM that allows the application to be installed. We recommend that you set this to the Visual FoxPro minimum, which is 64MB.

The screen resolution and color depth options are very personal settings. We have known users over the years who refuse to move past 640x480 no matter how large a monitor they use. Restricting the screen resolution should be negotiated in advance since there are laws that regulate accessibility issues in many countries.

How do I have the install files registered for all users of the computer?

There is a quirk in the initial release of InstallShield Express – Visual FoxPro Limited Edition that does not automatically load the installed application for all users on Windows. This happens if the installation does not use the Customer Information dialog (see **Figure 9**).

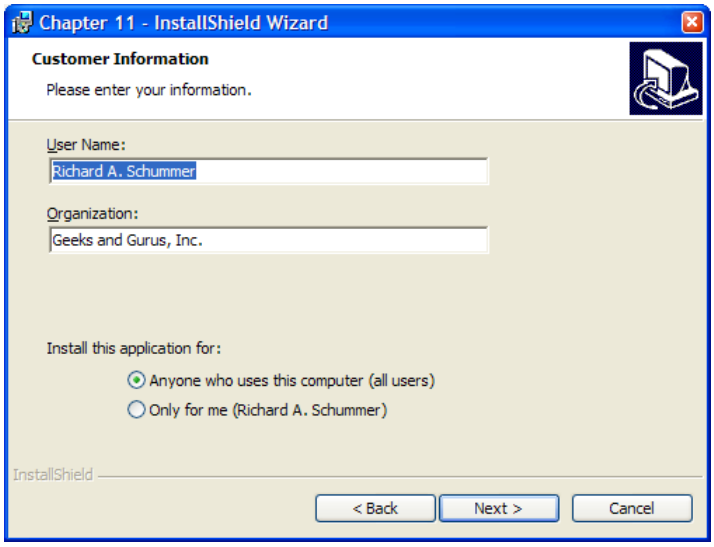


***Figure 9**. The Customer Information dialog allows the user to determine if the install is completed for all users on the computer or just for the user who installs it.*

The natural workaround is to include the Customer Information dialog in the install routine. If you run into installs that were built and shipped and it is not cost effective to re-release the application, you can still work around this issue by adding a parameter to the SETUP.EXE:

```
setup.exe /V"ALLUSERS=1"
```

There are three values that can be set for the ALLUSERS property. ALLUSERS=NULL (default value) will install the package for the current user. ALLUSERS=1 will install the package for all the users on the machine provided the user has administrative privileges. If the current user running the setup on Windows NT/2000/XP does not have admin privileges, then the setup will error out and abort. ALLUSERS=2 will check the user's privileges to see if they have admin rights. Pending the outcome of this check, it will install for all users if the user has enough admin privileges, otherwise just install it for the current user.

There are a number of excellent tips like this one available on http://support.installshield.com/kb/, http://support.microsoft.com/kb/, and http://fox.wikis.com/.

Visual FoxPro 6 Setup Wizard tips

We realize that Visual FoxPro 7 has been available for quite some time, but there are valid reasons to continue development in Visual FoxPro 6 and to leverage the Visual FoxPro 6 Setup Wizard.

One reason we still use Visual FoxPro 6 for a project is for applications where our customer has requested a minor enhancement to an application that is distributed to numerous sites or to one site with many workstations. Upgrading these projects to Visual FoxPro 7 would require each application to go through intensive system testing, and the redistribution and installation of the runtimes.

The Setup Wizard is not fancy. It has basic features, and is a 16-bit, run of the mill, not very flexible installation setup program. There are several commercial installation packages like Wise InstallBuilder and InstallShield that provide more flexibility and have more complexity. Microsoft has a product called the Visual Studio Installer available for the cost of a download from the Microsoft Visual Studio Web site. All of these packages provide a scripting capability so you can take control over the installation at the micro level. The Visual FoxPro Setup Wizard gives you few choices, provides the users with a simple interface, yet remains effective for the typical installations we assemble for customers. Would we recommend it for a vertical market application? Not likely. We note this only because it does not provide the micro control we would desire in shipping to clients with unlimited configuration combinations.

How do I run the Visual FoxPro 6 Setup Wizard?

The Visual FoxPro Setup Wizard is a Visual FoxPro application (APP). It is accessed via the Tools | Wizards | Setup Wizard menu option, through the Wizards Selection dialog (Tools | Wizards | All), or directly running it via the `DO (HOME()+"Wizards\WzSetup.app")` in the

Command Window. Unfortunately, the Setup Wizard source code is not available with the rest of the wizard source code that is distributed with VFP. You cannot really use the Setup Wizard with Visual FoxPro 7 since it is only smart enough to know about the Visual FoxPro 6 runtimes and components. Alternatively, you can copy the Visual FoxPro Runtime DLLs and direct them in the files section (step 6 of the wizard) to be loaded in WinSys directory.

How does the Setup Wizard retain its settings for the next build?

Visual FoxPro will read the last configuration of the last setup that is created when you start the wizard. It accesses the WZSETUP.INI file that was created by the last setup creation. Each item selected during the execution of the wizard is saved in the Preference section of the file. Some of the settings are obsolete, like the "Make1.2MegDisk", which was available in a previous version of the wizard.

Here is an example of the contents of the WZSETUP.INI:

```
[Preferences]
DistributionDirectory=C:\VFP98\DISTRIB\
DistributionSourceDirectory=C:\VFP98\DISTRIB.SRC\
SourceDirectory=D:\DEVVFP6APPS\HACKFORM3\
InstallFoxProRuntime=Y
InstallFoxProMTRuntime=N
InstallGraph=N
InstallHelp=N
InstallODBCDrivers=N
AccessDriver=Y
FoxPro2xDriver=Y
dBASEDriver=Y
ParadoxDriver=Y
SQLServerDriver=Y
ExcelDriver=Y
TextDriver=Y
OracleDriver=Y
Oracle7Driver=Y
BtrieveDriver=Y
VFPDriver=Y
InstallWindows95=N
InstallWindowsNT=N
DestinationDirectory=C:\Tools\
Make1.44MegDisks=N
Make1.2MegDisks=N
Make720KDisks=N
MakeNetsetup=Y
MakeWebsetup=N
SetupBanner=RAS HackForm
Copyright=Rick Schummer\n1997-2000
PostExecute=
UserDefaultDirectory=\HACKFORM3\
ProgManGroup=Visual FoxPro Applications
UserCanModify=1
SplitSize=363520
FileCustomizationDelimiter=~
InstRemoteAuto=N
InstActiveX=N
InstOLEControl=N
```

```
MakeDependencyFile=Y
MakeWebExecutable=N

[File Customizations]
HACKFORM.EXE=AppDir~YES~RAS HackForm~%sHackForm.exe~d:\hackform3\wrench.ico~NO
```

As you plug through the different steps of the Setup Wizard you will recognize each of the settings and how they relate to the items on the various pages. This obviously makes the second and subsequent runs much easier for the developer. Otherwise, you would need to remember each of the settings every time you build a new installation.

The Setup Wizard also requires two directories to be present under the Visual FoxPro Home directory. The first is the Distrib.src directory. This directory contains all the different base components and setup executable that come with Visual FoxPro and can be distributed as part of your application installation. This directory is created during the installation of Visual FoxPro and is required to run the wizard. The other directory is the Distrib directory, which contains setup configuration files created and populated during the current run of the wizard. The Setup Wizard can create this directory if it does not exist (see **Figure 10**), or you can locate it if you have a custom one located elsewhere on your local client PC or network drives.

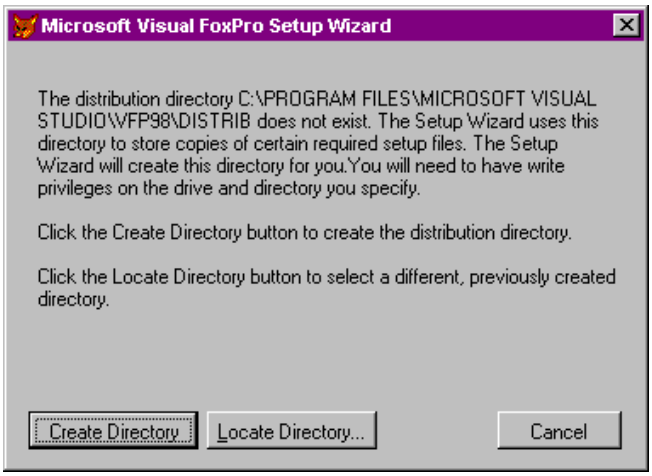


***Figure 10**. Visual FoxPro's Setup Wizard will prompt you to locate or create the needed Distrib directory.*

Each of the service packs that delivered updates to Visual FoxPro 6 had enhancements and/or bug fixes to the Setup Wizard. For instance, Service Pack 3 (SP3) delivered the new option of including the multi-threaded runtime files. SP3 also fixed a number of serious bugs that were in the original release of Visual FoxPro 6. We address several bugs throughout this chapter and have included a section on issues and bugs at the end. If you are using Visual FoxPro 3 or 5, check the Microsoft Visual FoxPro Web site for Setup Wizard updates in the download area. Microsoft released a number of updates to the Setup Wizard though this easy access mechanism.

What tips are there for Step 1: Locate Files?

Step 1 of the wizard is to select the root of the directory tree. If you pick the wrong tree you will likely not know it until Step 6 when you see the entire list of files that are going to be included. Pick the ellipses button to select a different directory if one is already specified or no directory is present at all. Don't be confused by the fact that the listed directory is in a read-only textbox.

The Setup Wizard works with only one installation directory tree at a time. We only include the files that are loaded at the customer site. Since we do not ship source code as part of the executable installation, we build a separate directory tree ahead of time and copy over the executable files. This needs to be completed before starting the Setup Wizard. The Setup Wizard will not recognize added files to the distribution tree until it is restarted. Deleting files from the distribution tree will cause the Setup Wizard to fail with a cascade of errors.

What tips are there for Step 2: Specify Components?

Step 2 is one of the more complicated steps as far as choices presented and deciding what to include in the customer's installation. Earlier in this chapter we discussed various strategies on what components are included on certain installs.

Here are some questions that you need to review before deciding what options are to be included in the installation. Are all of these components included with the installation (Workstation Install vs. Server Install)? Will the ActiveX controls we are including work with this step? Should I include the Visual FoxPro runtimes or put them on a separate installation? Do I need Microsoft Graph or ODBC Drivers? What about ODBC DataSources that are not handled by the Setup Wizard? Do I include the Help engine for future use?

A decision we have found beneficial is to include the Visual FoxPro runtimes on every first-time install for a customer. Whether the application files are loaded to the network or it is a WorkStation Install, include the runtimes because they are used by all Visual FoxPro apps. We are careful to know which version of the runtimes is needed. If we have a new feature that includes the Session object, we know we need to make sure that the customer is minimally running the Visual FoxPro 6 Service Pack 3 runtimes. We don't automatically ship runtimes because different customers are running different versions. Having a mix at a customer site sounds like too many support calls waiting to happen.

We do not automatically include the HTML Help Engine files unless we have a Help file to deliver. The HTML Help engine has evolved over time and new versions are probably in the works as you read this. The one you want to ship is the latest one when the Help file is ready. ActiveX controls are weird beasts; some work via this step, others require the method we used prior to Visual FoxPro 6, which is to copy the OCX file to the distribution directory and handle it in Step 6. The same goes for a COM object that you are including in the installation.

What tips are there for Step 3: Create Disk Image Directory?

The third step in the process determines where the install images are stored and what install images are generated. You can select one, two, or three images to be generated in one pass of the Setup Wizard. Which options to select will depend on your client's requirements and infrastructure.

We have not generated diskette install versions in a couple of years, ever since we bought an Iomega Zip drive. The basic Visual FoxPro hello world install was four or five diskettes, maybe more. The whole diskette installation seems like ancient technology since the advent of CD-Rs, CD-RWs, and super disks. If this option is still one that you need to build, be prepared to copy a number of files to a number of floppies. The nice thing is that the diskettes are broken down into subdirectories. One mistake we have made with the diskette option is to copy the "directory" (that is, DISK1) to the floppy. Only copy the contents of the directory to the floppy. The directory takes up bytes on the floppy and can make it so all the needed files will not fit on the media. It will also impede the install since it will not find the files in the root directory. There is a Microsoft KnowledgeBase article that also indicates that there is a problem with the Wizard leaving significant space open on each floppy, which bloats the number of floppies needed. See "Q191684: Setup Wizard Leaves 200KB of Disk Space on 1.44 Floppies" on http://support.microsoft.com for more details.

The basic WebSetup and the NetSetup both generate files in one directory. This includes the needed SETUP.EXE and supporting files. Both setups generate a single cabinet (CAB) file. The difference between the two is that the WebSetup process compresses the CAB file. The general concept is that the WebSetup install files will be used to transfer files via a low bandwidth mechanism like a dialup connection or can be used to handle a larger install set on smaller media. The NetSetup files can be copied or installed over a high-speed network or from a CD-ROM.

All this can be confusing, and the reality of the situation is as follows. If the non-compressed setup files fit on the media we are distributing, we use the results of the NetSetup. Otherwise, we use the WebSetup files. To date we have not had a release that will not fit on a single 650MB CD-R. Before we purchased the CD burner we used 100MG Zip disks. If the release would not fit uncompressed, we used the compressed files.

Either set of files can be copied or burned to the root directory of the desired media. You can also copy them to a subdirectory to include more than one install on the CD. This is how we burn the Workstation Install discussed in a previous section of this chapter. One directory is for the current Visual FoxPro runtimes, a second for the current Visual FoxPro multi-threaded runtimes, the third is the HTML Help Engine, the fourth is the Visual FoxPro ODBC Driver (and others if needed), and the last is the standard package of ActiveX controls. Change to the desired directory and run the SETUP.EXE. Naturally, if you are distributing the install as a download on the Internet you will want the WebSetup (and check "Generate a Web executable file" in Step 7) to have a single compressed file to download.

One error we have discussed on the online forums that is related to this step is "Error generating cab files: Error code 3." This error happens if you leave the Project Manager open and have the Setup Wizard disk images directory built to the same directory that the project distribution files reside (directory selected in Step 1). This error was fixed in VFP's Service Pack 3 and is something we thought you should be aware of if using an older version.

What tips are there for Step 4: Specify Setup Options?

This step allows you to customize the look and feel of the installation process. I always make sure that the company gets the full marketing plug from this step in the installation. Note that **Figure 11** displays the initial installation form and the About form (displayed via the Installation Control menu accessed by the icon in the upper left corner). The Setup Dialog

Caption is used in several places on this form. The copyright information is used only in the About form.

The optional post-setup executable is great if you need to execute some code just after the files are installed on the computer. This is an excellent way to run some code that updates the Windows Registry, or dynamically sets up a configuration file that point to various components in the application, or set up a desktop shortcut.

One interesting item we crossed in the Microsoft KnowledgeBase is "Q271405: Batch Files Do Not Run as Post Executables in Setup Routine for Windows NT 4 and 2000." Using batch files is pretty common for post-setup executables to copy files, create desktop shortcuts, and fire off a data conversion or data model update process. Unfortunately there is no suggested workaround for this problem.

It would be nice if the package also allowed items like a ReadMe text file so we could include some last-minute instructions and a custom license agreement, but these are the types of features offered in the commercial packages. The post-setup executable could also be used to run a file viewer on the ReadMe file if one is included.

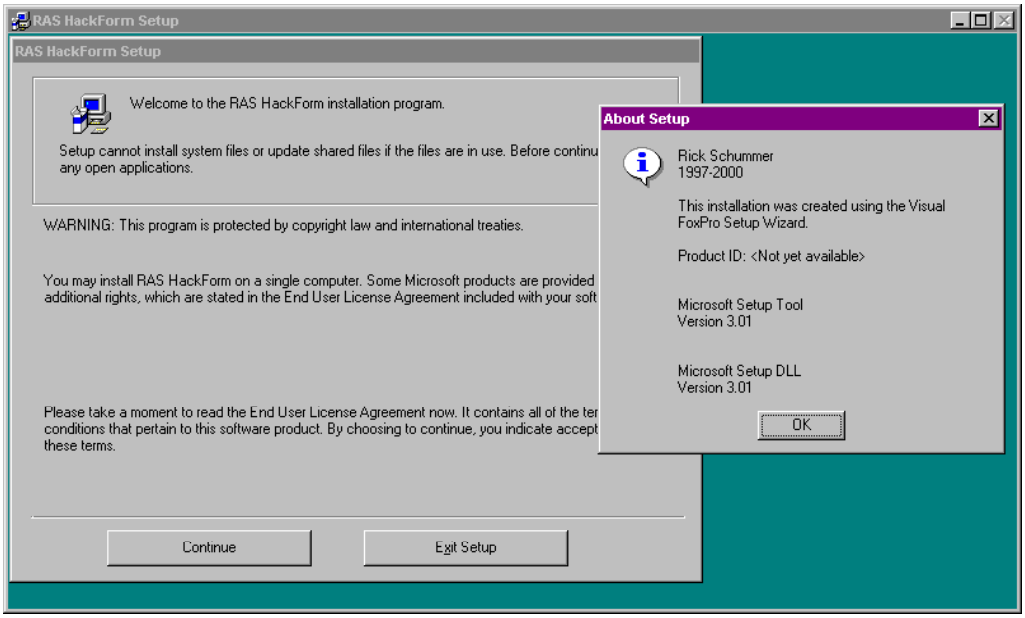


***Figure 11**. Items that are entered in Step 4 show up on the initial Installation dialog with the About Setup window.*

What tips are there for Step 5: Specify Default Destination?

The Default Destination step is important to provide flexibility to the users. We never hard-code a drive letter because it is ignored by the setup. The installation will always start on drive C and the user will need to switch it to a network or another drive if necessary.

The other two options provide important functionality for the user and flexibility that we demand from the software we install. Create a Program Group that translates into the menu

group added to the Windows Start | Program menu. Be creative in this respect. We like to utilize this option for the company name so all the company's suite of applications are accessible via one menu grouping. We always recommend that the user be allowed to change the grouping and the directory name. It is their computer, and it is their custom software that we are loading. On the other hand, the company you are developing for might want a standard enforced for the Program Group. The end users will always be allowed to change the directory.

What tips are there for Step 6: Change File Settings?

The Change File Setting step is the only place you will find a complete list of files that is going to be part of the install process. We tend to jump ahead to this step just to review that we have the proper directory selected in Step 1.

Program Manager is another old term from Windows 3.1 that seems to remain in the latest revision of the Setup Wizard despite the fact that we cannot build Visual FoxPro apps that run on Windows 3.1. The Program Manager (PM) selection defines which files are added to the client's Start menu (see **Figure 12**). Once you select any file to be a PM item you need to give it a description and the command line for that item. This translates to the item caption as it is on the menu, and the command line that would be in a shortcut. You can include parameters on the command line, just like you would in the Windows Start | Run menu. One important item is to include the %s at the beginning of the command line so that the installed directory is appended during the installation setup. This allows the user who selected their custom directory to also have it included on their menu.

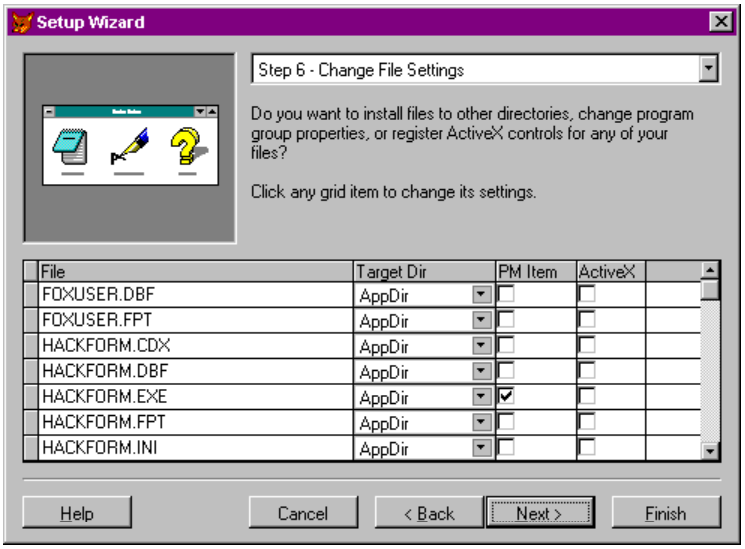


***Figure 12**. Defining the files that are on the Windows Start menu.*

If you have ActiveX components that were not included in Step 2, copy them to your installation directory (in advance). Then mark them in this step to be copied and registered as an ActiveX control. We want to remind you that Microsoft recommends handling all the

ActiveX files in the same manner. Make sure you use either Step 2 or Step 6 to process all the ActiveX files together. Over time we have used non-Microsoft ActiveX files like DynaZip, the Adobe FDF Toolkit, and the Amyuni PDF driver via Step 2. We have found from time to time that some of the Microsoft controls like the Common Dialog control need to be handled in this step. We're not sure exactly why, but it might save you some time if you run into this problem. Make sure to select the WinSys Target Directory for your ActiveX controls that need to be loaded in the Windows System directory.

What tips are there for Step 7: Finish?

Obviously the Finish page is where you decide to initiate the actual install build process (see **Figure 13**). At this point you get to decide if it is a go or no-go. There are only a couple of decisions left. One is to determine whether you want the dependency file to be created.

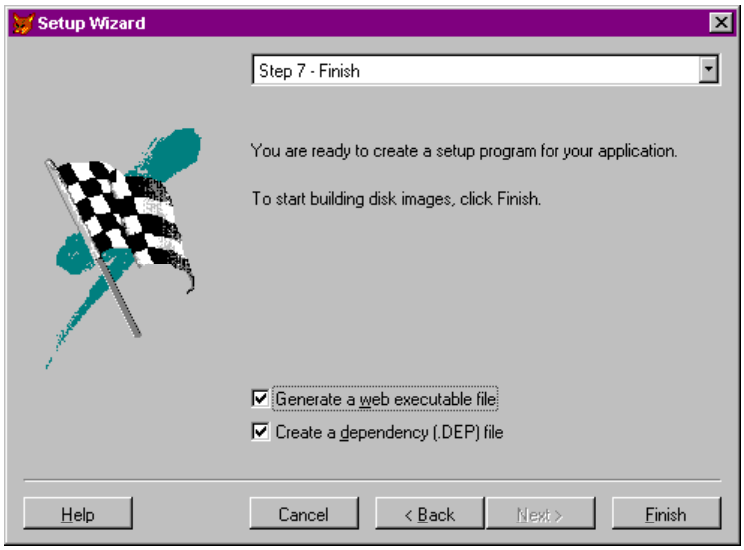


***Figure 13**. The finish line is reached!*

The generation of a Web executable file option is only available when the WebSetup disk images option is selected in Step 3. If this option is selected, one file is created (WEBAPP.EXE) so only one file needs to be downloaded from a Web site. This is a common source of confusion with Visual FoxPro developers. Most developers think selecting a WebSetup builds an install for the Web. Selecting the WebSetup disk image only compresses the setup files in one CAB file, but still has several other files that are part of the setup. If the single file option is selected an additional process is spawned in a DOS session to build the single executable. The WEBAPP.EXE is not located in the WebSetup directory, but in the base directory picked in Step 3.

The WZSETUP.INI is generated before the installation images are created so the same settings can be used when building the installation for the same distribution directory the next time the Wizard is run.

How do I AutoRun Visual FoxPro 6 installations?

The professional shrink-wrap applications that we install on our development computers typically execute by themselves when the CD-ROMs are loaded in the CD drive. This is a feature called AutoRun, and it is standard on the Windows 95/98/ME and Windows NT4/2000/XP operating systems. The AutoRun feature automatically detects the CD when it is inserted into the CD-ROM drive and runs an application based on the contents of the AUTORUN.INF file located on the CD-ROM. The standard Visual FoxPro 6 Setup Wizard does not build this file, so Visual FoxPro developers have to take additional steps to add this functionality to our application installations.

The AUTORUN.INF file must reside in the CD's root directory. The basic layout of the AUTORUN.INF file is similar to your basic configuration INI files. It has a Key section and properties settings. Here is an example:

```
[autorun]
OPEN=AUTORUN.EXE
ICON=WRENCH.ICO
```

The OPEN entry indicates the location of the AUTORUN.EXE file. The file is assigned to this property setting with a specific or relative directory. This means that the program that is automatically executed can reside in any directory on the CD. The ICON entry indicates the icon file used to represent the CD-ROM drive in the Windows Explorer (see **Figure 14**). This icon is displayed on the Address line and in the TreeView.

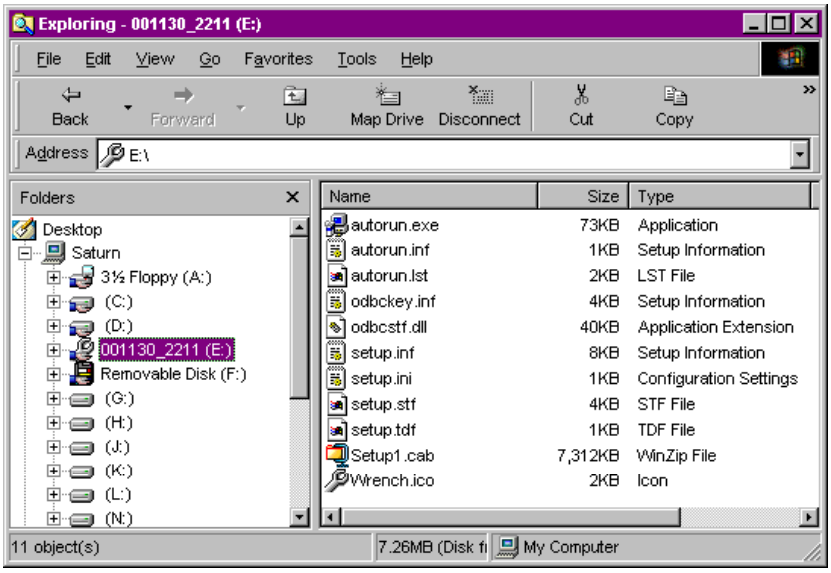


***Figure 14**. Using the ICON setting in the AutoRun.inf file provides control over the icon used for the CD-ROM drive.*

This is an excellent way to automatically display the ReadMe in HTML format when the install CD is loaded. To accomplish this you need to shell an HTML document.

```
[autorun]
OPEN=ShelExec index.htm
```

To create an AUTORUN.INF file that launches a Visual FoxPro setup routine, you must first create the distribution set using the Setup Wizard. There is one file that needs to be renamed and another one copied once the Visual FoxPro setup files are generated. First make a copy of the SETUP.EXE, and rename it to AUTORUN.EXE. You need to copy this file because the Setup routine still looks for SETUP.EXE during the setup process. Next you need to rename SETUP.LST to AUTORUN.LST. Last, create a new text file named AUTORUN.INF with the following contents:

```
[autorun]
OPEN=AUTORUN.EXE
```

Burn these files from the distribution set to a CD. What is next? Verify that the CD works! We like testing; it makes sure that we look good in front of the clients. The simple test is to insert the CD into the CD-ROM drive on a different computer if possible. Optimally this is a computer that has no Visual FoxPro installs (which might be difficult to find in a Visual FoxPro development shop). The application's setup should be launched soon after the CD starts spinning. Install the files and run the entire setup to a successful completion. This is the ultimate test. We have an older computer in the office that has Norton's CleanSweep loaded. This allows us to completely remove the test load once we verify that all the runtimes, ActiveX controls, and executables are loaded and registered properly.

One thing that you need to know is that you will have to follow the steps to copy and rename files each time you do a build. The setup files are erased each time a new setup is created. You might want to save the AUTORUN.INF file off to another directory or build separate directories each time.

What are the additional setup parameters?

Most setup executables are run via the operating system AutoRun mode or via the Windows Start | Run dialog without parameters. There are a number of parameters that are available via the standard Visual FoxPro setup that developers can leverage for their custom applications. These optional switches (also known as parameters) can be displayed by sending the invalid "slash" to a Visual FoxPro setup. The setup will display an error message first and is followed next by the Setup – Usage dialog (see **Table 3** and **Figure 15**) that lists off all the supported install switches.

***Table 3**. Setup optional parameters.*

Parameter	Activity affected
/A	Administrator mode
/G filename	Generate logfile of installation activity
/Q[0|1|T]	Quiet install mode (0 shows exit, 1 hides exit, T hides all display)
/QN[1|T]	Quiet install mode with reboot suppressed
/R	Reinstall the application
/U[A]	Uninstall the application but leave shared components (/UA to remove all)
/X filename	Set Network Log Location for tracking install instances
/Y	Install without copying files

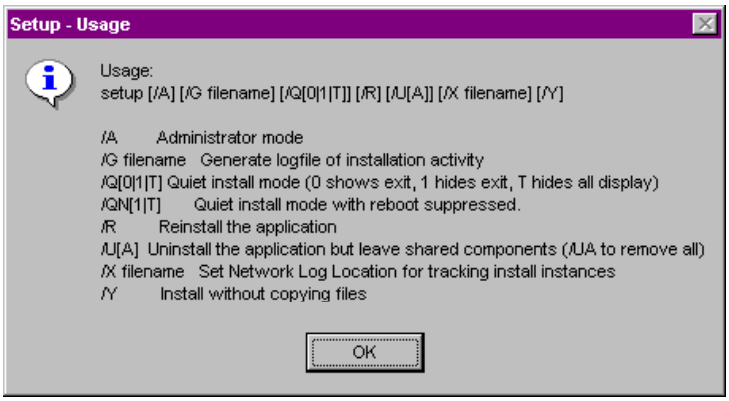


***Figure 15**. A message is displayed when an invalid switch is sent to a Visual FoxPro executable.*

How do I get a list of files and changes from the install?

If there is one thing we like about top gun installs, it is to know what a product loads and modifies on our computer. The /G switch accomplishes this and does a very thorough job recording the setup settings, the command line of the install as it was executed, the date and time it was run, the Registry entries that it added, the directories it created, the shortcuts it generated, and the files it copied. We are impressed with this feature as developers. It has been helpful during installation support calls when files appear not to get installed or registered.

```
E:\setup.exe /G c:\temp\custominstalllog.txt
```

The install will fail if the directory specified for the log file does not exist or if it cannot be created because of space or security limitations

How do I have a user reinstall an application?

Ever had users delete some key component to the application just before that critical month end batch process needs to be fired off? Have them reinstall the application via the /R switch. This switch will reinstall all the files again. Be careful if you include a mechanism to install data as part of the setup because it could accidentally reinitialize all the applications tables.

How do I have a user uninstall an application?

As if any customer would want to uninstall our most excellently crafted applications, the option to remove all the files is available by using the /U switch. This switch can also be used in conjunction with the /G switch to log all the file and Registry key removals.

```
E:\setup.exe /U /G c:\temp\custominstalllog.txt
```

You can use the /UA switch to remove all the application files and any shared components that were loaded during the initial installation. This feature is helpful when you test the install before shipping it to a client.

How do I have a user install without intervention?

At times, you may wish to have your Visual FoxPro application install without user intervention. You can accomplish this by using the Quiet install mode with the SETUP.EXE. There are several switches that will produce the Quiet mode operation. By using this option, the user has no dialogs like the registration name and organization, directory selection, or selecting the Program Group.

```
Setup.exe /Q or /Q0
```

When you execute the installation using either of these switches, Setup opens a dialog box with "Initializing Setup..." and the user sees the progress bar of the current file copy status. When the setup is finished, the dialog box is displayed indicating whether the setup completed successfully. This message requires the user to click an OK button to complete the installation.

If you want to eliminate the message that tells the user whether the install was successful or not, but still displays the progress meter, try the following command:

```
Setup.exe /Q1
```

This next setup command line option does not display a single window or dialog box. This is the ultimate stealth install. The user will never know whether the installation happened, and more importantly, will never know whether it succeeded or failed. This option might be handy if you are installing some optional features or maybe including some Microsoft components that update the operating system after the application installation.

```
Setup.exe /QT
```

The "N" option on the /Q switch will suppress any needed reboot that might be initiated by the files that are loaded as part of the install.

```
Setup.exe /QNT
```

While the concept of installing an application in complete stealth mode seems a bit radical, the ability exists for flexible installations.

How can I create a desktop shortcut using the Setup Wizard?

(*Example: CH11.vcx::cusShortcut*)

One of the limitations of the Setup Wizard most addressed on the various online forums is how one can create a shortcut on the desktop. The Setup Wizard only allows a shortcut to be created on the Start menu.

One method is to write a program that copies the Start menu shortcut. This code would need to search the drive for the LNK file. The drawback of this technique is that you are still limited to the default properties set up for the shortcut.

The operating system exposes the functionality to create shortcuts via the Windows Scripting Host (WSH). There are a number of properties you can control for a shortcut if you leverage the WSH that are not available using the native Start menu option included in the

Setup Wizard. These properties include the folder that the shortcut is generated in, a hot key to assign to the shortcut, a description, and the window state when the application is started.

There has been a rash of viruses like the ILOVEYOU virus that use the Windows Scripting Host to delete and rename files. These viruses caused many system administrators and end users to uninstall this Windows service. This can cause tools like the shortcut creator code described in this section to fail.

Here is an example of a program that calls a class included in the book downloads that creates a shortcut on the Windows desktop to the Visual FoxPro 7 development executable.

```
SET CLASSLIB TO ch11.vcx
loShortcut = CREATEOBJECT("cusShortcut")

loShortcut.cSpecialFolder    = "Desktop"
loShortcut.cTargetPath       = "C:\PROGRAM FILES\VFP7\VFP7.EXE"
loShortcut.cWorkingDirectory = "D:\DevVFP7Apps"
loShortcut.cHotKey           = "Ctrl-D"
loShortcut.cDescription      = "Best application developer tool"
loShortcut.cArguments        = "-t"
loShortcut.cFileName         = "VFP 7 Rocks"
loShortcut.nWindowStyle      = 1
loShortcut.Create()

?"Failure Message is = ", loShortcut.cFailureMessage
```

Here is a partial listing of code in the shortcut class creation method:

```
* cusShortcut.Create()
* Start the Windows' Scripting Host
loWSHShell           = CREATEOBJECT("wscript.shell")

IF VARTYPE(loWSHShell) = "O"
  lcSpecialFolder = loWSHShell.SpecialFolders(this.cSpecialFolder)

  IF NOT EMPTY(lcSpecialFolder)
    loShortcut = loWSHShell.CreateShortcut(ADDBS(lcSpecialFolder) + ;
                                           this.cFileName)

    IF VARTYPE(loShortcut) = "O"
      WITH loShortcut
        .TargetPath       = this.cTargetPath
        .Arguments        = this.cArguments
        .Description      = this.cDescription
        .IconLocation     = this.cIconLocation
        .Hotkey           = this.cHotkey
        .WindowStyle      = this.nWindowStyle
        .WorkingDirectory = this.cWorkingDirectory
        .Save()

        llReturnVal = .T.
      ENDWITH
    ENDIF
  ENDIF
ENDIF
```

There are a number of special folders that developers can use to create shortcuts in using the Windows Scripting Host:

- AllUsersDesktop
- AllUsersStartMenu
- AllUsersPrograms
- AllUsersStartup
- Desktop
- Favorites
- Fonts
- MyDocuments
- NetHood
- PrintHood
- Programs
- Recent
- SendTo
- StartMenu
- StartupB
- Templates

How do I find out about Setup Wizard issues and bugs?

Naturally, we want to believe that each version of Visual FoxPro is better and that Microsoft has released fewer and fewer bugs with every version. Doing a query on Microsoft's KnowledgeBase for Visual FoxPro issues with "kbAppSetup kbVFp600" as text to search for will bring up a number of KnowledgeBase article that note fixes in the Visual FoxPro 6 Service Packs as well as various issues that are still outstanding and some of the workarounds that are suggested.

How can I ensure a smooth deployment?

We have spent years developing and deploying applications. While there is no perfect checklist or plan to successfully deploy applications, there are several items to note that definitely lead to a smoother implementation. Planning up front (even as soon as the requirements collection phase of the development life cycle) is the key.

Duplication

Once the software is ready to ship, you have to employ a mechanism to get it to the sites for deployment. Will you have the users download the setup routine from the company Web site? Will you burn a set of CDs and mail them through the postal service or though one of the many overnight carriers?

The introduction and vast acceptance of the Internet has eliminated some of the duplication for software deployment. Skip this detail if you are going to distribute via a Web site.

Are the CD burners ready to roll? They are so cheap these days that there is very little excuse not to have one. They save time if you are doing a mass installation. We really dislike doing floppy installs since it takes so much more time. There are service providers that will duplicate CDs if you do not have access to a CD burner or have a mass distribution (anything over 200 CDs may be worth the cost). There are several CD label makers that will add that last-minute polish to the distribution. Whether you do floppy or CD installations, make sure you have enough media on hand to cut the installations.

Users

Our experience has proven that the clients/customers we develop the application for are typically not the ones who are going to be using the product. It is important to keep information flowing to the actual user base to keep them updated about the upcoming release.

Communication allows them to get training scheduled, have the equipment installed, cut the check to pay for the package, and schedule the parade in your honor for making life easy in the business world. Seriously, there can be a lot of work preparing the marketing literature, changing office procedures, and updating Web sites. Make sure you are communicating with the user community, either directly or through your customer contacts. Make sure to track e-mail, phone calls, and possible visits to their site(s) if needed.

Hardware

How many times have you shown up with the CD (or diskettes, tapes, Zip disks) to find out that the special label printer they need for the application was never ordered? Have you shown up with a professional-looking CD just to find out no machine in the office has a CD player because the boss does not want them used to play music in the office? Many custom apps need new hardware. Whether it is the latest in Pentium technology or the simple fact of dumping the dot matrix printers for a 32-page-per-minute laser printer, many applications have special hardware needs. Verification that needed hardware is delivered ahead of the application can save some embarrassment in the delivery of your new functionality.

Training materials

Training materials may be required for broad released applications or vertical market applications. Many customers we have worked with in the past write and develop the training materials for their applications. We like this concept since it gives them ownership in the process. It also saves them plenty of cash and allows the development teams to concentrate on what they do best.

Conclusion

There are a number of options when creating a deployment strategy. In this chapter we covered different techniques for extracting version information from the executables, different strategies for developing install packages, and covered the two different install builders included with Visual FoxPro and some tips and gotchas when working with these tools.

Chapter 12
VFP Tool Extensions and Tips

Visual FoxPro ships with a number of tools to enhance the development experience. These tools come in the form of designers (forms, classes, databases, tables, reports, and menus), those described by Microsoft as the XBase tools (Class Browser, Component Gallery, Object Browser, Task List, IntelliSense Manager, and Coverage Profiler), and the wizards and builders. These tools can all be expanded, extended, enhanced, or even replaced to provide new functionality or improve the tool's usability.

The focus of this chapter is to provide tips and tricks when using the various tools provided with Visual FoxPro by Microsoft as well as some handy extensions and add-ins for these tools. We will only address the Menu Designer, Coverage Profiler, Task List Manager, Object Browser, and Project Manager in this chapter.

Menus

The Menu Designer has been enhanced in Visual FoxPro to provide Top-Level form menus, shortcut menus, and in Visual FoxPro 7 the capability to display icons in the menus. Other than these three additions the Menu Designer in Visual FoxPro is for all practical purposes the same since the days of FoxPro 2.5.

The Visual FoxPro menus are modified via the Menu Designer and the menu source is stored in the metadata file (DBF/FPT with a MNX/MNT extension). Visual FoxPro does not use the metadata directly in your application like it does for forms, classes, reports, and labels. It first requires that a program be generated from the menu metadata. The metadata is translated into a program during a project build, or via the Menu | Generate menu option. This process uses Visual FoxPro's GENMENU.PRG to generate the program code. The resulting menu code is stored in a program with the MPR extension.

How can I dynamically change captions in menu? (*Example: MenuChapter12Example.mnx/mnt*)

A Visual FoxPro developer typically will hard-code the menu bar prompts in the Menu Designer. Developers who work with applications that run in multiple languages, or have requirements to build menus that have the captions change dynamically, need a different approach to display and create captions that can change.

The approach we will use is to call a function in the menu prompt. The example menu has two menu bars on the Help pad that demonstrate this technique (with three items dynamically set, two prompts and one message). We added a simple example procedure in the menu Cleanup called MyApp. The function accepts a character string using the new Visual FoxPro 7 **INPUTBOX()** function to demonstrate the dynamic characteristics.

```
FUNCTION MyApp()
* This function could easily call an application
* level service which returns the application caption
```

```
LOCAL lcAppCaption

lcAppCaption = [My Killer Custom Application    ]
lcAppCaption = INPUTBOX("Input your dynamic caption...", ;
                                "Dynamic Caption Example", ;
                                lcAppCaption, 5000, lcAppCaption)

RETURN lcAppCaption
```

This demonstrates how your menu can use the application name in the prompts. You are more likely to make a call to a system or application level service that returns the caption for the prompt or menu item messages. Another option to use the application name might be to simply return the screen.Caption property. The menu bar prompts in the sample menu are as follows:

```
" + ALLTRIM(MyApp()) + " on the \<Web
\<About " + ALLTRIM(MyApp()) + "..." + "
```

Now you might be looking at the prompt values and wonder why the quotation marks are unbalanced or look backward from a normal Visual FoxPro evaluation viewpoint. The point to remember is that these values are substituted in a MPR by GENMENU.PRG. GENMENU.PRG adds quotes around the prompts when it generates the code for the **DEFINE BAR** code. It takes some time to get use to the syntax, and might even take a couple of passes to work out the value that is needed when performing the dynamic string building for the prompt. Here is some of the code from the MPR that shows the two example menu bars and how they are generated:

```
DEFINE BAR _mst_vfpweb OF _msystem PROMPT ""+ALLTRIM(MyApp())+" on the \<Web" ;
  PICTRES _mst_vfpweb ;
  MESSAGE "Launches your web browser to go to Visual FoxPro's Web sites"

DEFINE BAR 7 OF _msystem PROMPT "\<About " + ALLTRIM(MyApp()) + "..." + "" ;
  PICTRES _mst_about ;
  MESSAGE "Displays version and copyright information about "+ALLTRIM(MyApp())
```

You can see that the first quote of **" + ALLTRIM(MyApp()) + " on the \<Web** is used to balance the inserted quote at the start of the prompt, not to include the evaluated **ALLTRIM()** in the string itself.

The dynamic menu approach included in this section will only be evaluated one time, when the MPR is executed. If you need constant changes to the menu you will need to rerun the menu program or provide a different technique that releases and adds menu bars to the menu. This is a limitation of the static nature of Visual FoxPro menus.

The example shows how to integrate the application name into the menu. The function call could just as easily call a class method, COM object, program, procedure, or function that does a table lookup to translate a string into a specific language equivalent. In essence, it can be any kind of code and can be anywhere that Visual FoxPro has access. The sample menu

also demonstrates that you can perform the same dynamic prompt technique with the menu bar message that is displayed on the status bar if it is active.

How can I permanently disable a menu option?

Menu bars can remain on the menu and be permanently disabled. There are two ways to accomplish this. Setting the SkipFor expression to .T. is the more obvious way is to make this happen. The other way is to start the Prompt with a "\" (without the quotes). This is an excellent way to disable a feature that is going to be included in a future release or temporarily disable a feature that has a defect when the rest of the release needs to be deployed.

How can I dynamically disable menu bars in menu? (*Example: MenuChapter12Example.mnx/mnt*)

Disabling menu items has long been a technique to disallow users from executing items that should not be executed at a particular time. One example of this is to disable the Cut/Copy menu items when no text is highlighted. There are several techniques available to Visual FoxPro developers.

First, we want to start out by discussing the mechanism that Visual FoxPro provides to disable a menu item. Each menu item has a **SKIP FOR** clause that is entered via the Prompt Options dialog (see **Figure 1**) of the Menu Designer. You can set a menu pad to be completely skipped (handy when you do not want any menu bars accessible) as well as individual menu bars. An example of this might be a navigation menu that provides users with the ability to move the record pointer to the first, previous, next, or last record. You only want that menu pad accessible if a form is opened that needs this functionality. To access the **SKIP FOR** clause you need to open up the Prompt Options dialog for the menu pad or menu bar. This clause must evaluate to true (**.T.**) or false (**.F.**). If it is true the menu item is disabled, and if it evaluates to false the menu item is enabled. The code generated by the menu generator will be something similar to this:

```
DEFINE BAR 6 OF _msystem PROMPT "\<About " + ALLTRIM(MyApp()) + "..." + "" ;
  SKIP FOR SkipForExample() ;
  PICTRES _mst_about ;
  MESSAGE "Displays version and copyright information about " + ;
          ALLTRIM(MyApp())
```

Every **SKIP FOR** clause on the menu is evaluated when the menu is activated. This can be demonstrated on the example menu MENUCHAPTER12EXAMPLE.MPR. There are two menu items that have calls out to a function called SkipForExample() (the View pad, and the Help | About option). This function displays a text message each time a skip for is evaluated. Clicking the mouse (or the first keystroke) that activates the menu makes two display lines. Once the menu is active the calls no longer are made. This can cause problems if the user leaves the menu active and the situation that the **SKIP FOR** accounted for no longer exists. Be careful what the **SKIP FOR** clauses evaluate to, and how that evaluation can change. Also note that the menu is "activated" when your application gains focus (via an Alt-Tab, or the user clicking on the application window). It also gets "activated" if there is code that adds pads, removes pads, or alters the menu in another way.

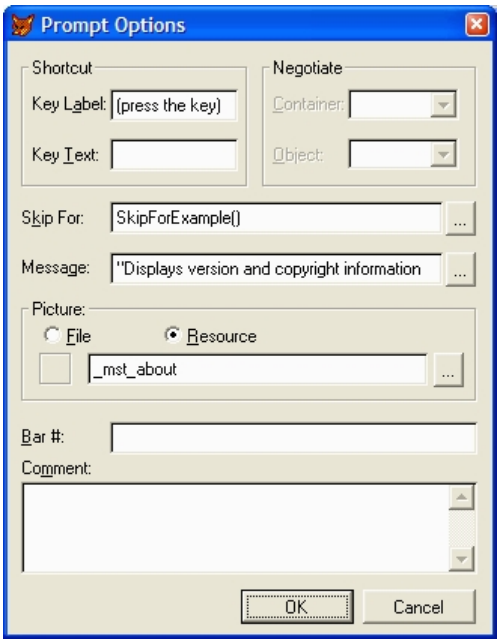


***Figure 1**. The SKIP FOR condition is found on the menu Prompt Option dialog.*

You can also permanently disable the menu item by placing a `.T.` in the **`SKIP FOR`** clause. Naturally this is not the most flexible approach to disabling menu options, but it actually can be handy for those features that are in process, but not quite complete for the release that needs to be made now. While it might be a good idea to remove the menu item completely, it can be important to leave it on to show the customer that you have not forgotten it, but it is not yet ready to test.

Another technique is the hard-coded public variable approach. It is not a technique that we promote, but one that you may encounter in a development project. The public memory variable (or a private memory variable scoped to the main program) must evaluate to true or false. We have seen developers alter the public variable in numerous places. The problem with this technique is that it is difficult to figure out the state of the variable at a certain point in time, and worse, exactly which code fired that set the public memory variable. This technique can be very difficult to debug.

The most flexible approach is to call a function that returns true or false. This function can be a function in a procedure file or something as elegant as a method on a security object. The only requirement of the function is that it returns the true or false needed by the **`SKIP FOR`** clause.

There is one gotcha that we want to make you aware of—functions in menu Cleanup snippets likely will be out of scope in an application. So call a method in a security object or a function in a procedure file if you use this technique. We got bit by this even when trying to encapsulate the code for the examples for this section.

How can I remove menu pads and bars from a menu?

It is easy to disable menu options as we demonstrated in the last section, but the technique often leaves users asking the question—how come I cannot access this option? If the user is not supposed to access the feature because of security, why not just remove it from the menu? In this section we will demonstrate a couple of techniques to make menu bars disappear.

The first method is the brute-force method, which calls Visual FoxPro code that releases menu bars or menu pads.

```
RELEASE BAR 23 OF _medit
RELEASE BAR _med_link OF _medit
RELEASE PAD _msm_edit OF _MSYSMENU
```

The code to remove menu items can be placed in the menu Cleanup snippet. This technique is easily broken as soon as someone changes the name of the pad or popup and forgets to modify the Cleanup code. The maintenance of the menu quickly becomes a nightmare.

A better technique is to implement a public domain tool called GenMenuX. GenMenuX is a wrapper to the GENMENU.PRG. From the extensive documentation, GenMenuX provides Visual FoxPro developers with the following capabilities:

- The ability to control default menu positioning, colors, and actions without manually changing the MPR file.
- The ability to remove menu pads based on logical conditions instead of using **SKIP FOR**.
- The ability to automatically add hot keys to menu pads.
- The ability to call menu "drivers" at various points in the Menu Generation process.
- The ability to define Menu Templates that contain standard menu objects that can be inserted at any time into an existing menu.

There are numerous directives available with GenMenuX. It is not within the scope of this chapter to discuss all the features of this product.

To use GenMenuX you will need to change the **_GENMENU** system variable to point to the GenMenuX program. You can do this in the Tools | Options dialog on the File Locations page. The option to change is the Menu Builder. You can programmatically change the **_GENMENU** system variable from the Command Window or a startup program. The last option is to set the **_GENMENU** system variable in your CONFIG.FPW file:

```
_GENMENU = GenMenuX.prg
```

Directives for GenMenuX are placed in the comments section of the menu item. All the directives start with "*:". This tells GenMenuX that it has something special to process. The

directive needed to remove menu items is "*: IF <cExpression>". The reference to cExpression is some code that evaluates to true or false when it is executed.

GenMenuX was written by Andrew MacNeill and can be downloaded from www.aksel.com/genmenux and www.stevenblack.com/SBCPublicDomain.asp. Many of the commercial frameworks ship a copy of this tool with their products to leverage ideas presented in this section.

The GenMenuX directive logic is opposite of the `SKIP FOR` logic for the menu item because the generated code will have a `NOT` added to the beginning of the code. For instance, if you had a menu item disabled unless your user id was "administrator," in the `SKIP FOR` you would have something like `NOT ("administrator" $ LOWER(SYS(0)))`. To have the same menu item removed from the menu using a GenMenuX directive, you would code *:if `("administrator" $ LOWER(SYS(0)))`. **Table 1** lists some example directives and the resulting code that's generated.

***Table 1**. Example directives and the code that is generated by GenMenuX in the resulting MPR.*

GenMenuX directive	Generated code

`*:IF USED("LoginHistory")`	`IF NOT (USED("LoggedInUsers"))`
	`  RELEASE BAR 3 OF Admin`
	`ENDIF`

`*:IF !IIF(TYPE("oUser.BaseClass")#"C",`
`.t.,`
`oUser.oSecurity.GetAccess("MP_ADMIN")>`
`1)`
	`IF NOT`
	`(!IIF(TYPE("oUser.BaseClass")#"C",.t.,`
	`oUser.oSecurity.GetAccess("MP_ADMIN")>`
	`1))`
	`  RELEASE PAD _0lx06r3es OF _MSYSMENU`
	`ENDIF`

`*:IF FILE("instructions.pdf")`	`IF NOT (FILE("instructions.pdf"))`

The code generated by the directive is located in the Cleanup section. Therefore, the menu code is built and the release code is automatically generated for you, no extra maintenance needs to be done when you change the menu pad or reorder the menu bar items.

How can I create a menu to use as a template for my VFP apps?

(*Example: MenuTemplateWizard.prg*)

Visual FoxPro developers have been voicing displeasure that the native product does not support object-oriented menus. The thing that would be nice about object-oriented menus is that a developer could create a base class menu with menu options that are common to all their applications. Subclasses of this menu could be created and augmented for customer-specific menu options, and later subclassed for an application. Since we do not have this available in the product we have to resort to menu templates.

This is about as trivial a concept as we have as Visual FoxPro developers. Open up the Menu Designer and assemble a menu with the basic application level services. Our menu template includes the menu options shown in **Table 2**.

***Table 2**. Sample options for a menu template.*

Pad	Options
File	Close, Printer Setup, Logout, Reindexing, Exit
Edit	Undo, Redo, Cut, Copy, Paste, Clear, Select All, Replace
View	(None, but add forms custom to application)
Report	(None, but add reports custom to application)
Tool	Change Password, Options, App Administration
Window	Cascade, Arrange All, Hide All, Show All, Next Window (cycle)
Help	Contents, Search for Help On, Technical Support, About

The easiest way we know to start a menu template is to do a Quick Menu from the Menu menu pad available when the Menu Designer is open. Delete the menu options that are not of importance to your applications and add options that are not included in the Quick Menu. Save the menu to a common folder that is accessible to the development team.

When you start a new project, copy the menu template to the project folder and add it to the project. This will save you time each time you start a new project. You can even automate this process as part of a new project wizard type application with code that's shown in **Listing 1**.

***Listing 1**. Code from MenuTemplateWizard.prg, which could be part of an application wizard.*

```
LPARAMETERS tcProjectDir, tcProjectName

LOCAL lcProjectName, ;
      lcMainMenuName

#DEFINE ccMENUFOLDER   "d:\MyFrameWork\Common\Menus\"
#DEFINE ccMAINMENUNAME "MainMenu"
#DEFINE ccMENUTEMPLATE "MainTemplate"

tcProjectDir   = ADDBS(ALLTRIM(tcProjectDir))
tcProjectName  = ALLTRIM(tcProjectName)
lcProjectName  = tcProjectDir + FORCEEXT(tcProjectName, ".pjx")
lcMainMenuName = JUSTSTEM(ccMAINMENUNAME)

CREATE PROJECT (lcProjectName) NOWAIT SAVE NOSHOW NOPROJECTHOOK

IF TYPE("_vfp.ActiveProject") = "O" AND !ISNULL(_vfp.ActiveProject) ;
      AND FILE(lcProjectName)
  IF FILE(ccMENUFOLDER + ccMENUTEMPLATE + ".mnx")
    COPY FILE ccMENUFOLDER + ccMENUTEMPLATE + ".mnx" TO lcMainMenuName + ".mnx"
    COPY FILE ccMENUFOLDER + ccMENUTEMPLATE + ".mnt" TO lcMainMenuName + ".mnt"

    * Add Menu
    lcFile = tcProjectDir + "menus\" + lcMainMenuName + ".mnx"
    DO AddFileToProject WITH lcFile
```

```
  ELSE
    ? "  Menu Template Files not available for copying at " + TTOC(DATETIME())
  ENDIF
ENDIF

RETURN


PROCEDURE AddFileToProject(tcFileName)

IF FILE(tcFileName)
  _vfp.ActiveProject.Files.Add(tcFileName)
  ? "Added file: " + tcFileName + " at " + TTOC(DATETIME())
ELSE
  ? "  Problem adding file " + tcFileName
ENDIF

RETURN
```

Note that the constant ccMENUFOLDER will need to be changed to meet your configuration.

How do I programmatically execute a VFP provided menu bar?

All Visual FoxPro system menu items can be activated via the `SYS(1500)` function. This function can be handy if you want to control menu items programmatically. Unfortunately, the menu items cannot be custom menu items you add to the menu. This comes in handy when you want to execute the menu option without the user selecting it with keystrokes or menu clicks.

We are not sure how practical this is in a production application, but we definitely see some uses for developer tools. An example of how this might be used is to consider a tool that adds source code to the current editor. You can add text the clipboard via the `_ClipText` system variable. The editor can be activated by the ACTIVATE WINDOW, followed by a call to the paste menu option:

```
SYS(1500, "_MED_PASTE", "_MEDIT")
```

How do I include native VFP menu items in a custom menu?

Visual FoxPro developers might be familiar with the Quick Menu option that will add the standard development items to the menu you are creating. The problem with this method is that you need to start with a new menu and then have to cut and paste the menu items to the application menu you are developing.

The Menu Designer provides an Insert Bar button to add menu bars that have default functionality (see **Figure 2** for the dialog). Any menu option (even the ones that do not make sense in a production application) can be selected. This functionality can be added to system menus, shortcut menus, and top-level form menus.

There are some extremely helpful menu bars like the standard clipboard functionality (cut, copy, and paste), data entry object content manipulators (select all, clear), and even items that help users work with the contents of edit boxes (find, replace). One handy option that works

with forms that we like is the Close bar found on the File menu. Developers might be unaware that the macro editor is available for production applications as well as the development environment using the Macro bar. The options found on the Window menu like Cascade, Arrange, and Cycle are also useful in production apps. If you use a Help system in your apps you will find the Help Contents already hooked in.

One quirk with Visual FoxPro 7 is that the icon resource does not come along by default when you add the bar. Make sure to go back into the menu item options, select the resource, and then press the ellipsis button. This brings up the list of menu bars to pick from and inserts the icon resource into the dialog. No need to generate your own icons. These icons can also be used for your own custom menu items. Just make sure they make sense for the custom item since many of these icons are industry standard for items that are common to all Windows applications.

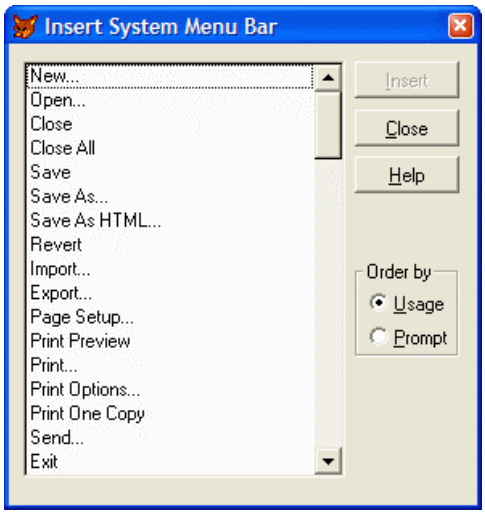


***Figure 2**. This is the dialog to select one of the Visual FoxPro provided menu bars.*

The default shortcut keystrokes are included when you select the menu bar to be included. You can change them if you like, but remember, many of the keystrokes are implemented with Windows standards in mind.

How do I create and implement a shortcut menu? (*Example: EditShortCut.mnx/mnt, ShortcutDemo.scx/sct*)

Shortcut menus have been around since Visual FoxPro 5, yet they remain a mystery to some developers. The implementation is straightforward: Create a menu via the `CREATE MENU` command, select Shortcut when prompted, and use the Menu Designer to create the single menu popup with as many menu bars as practical for the application. Shortcut menus can also have submenus.

The interface standards dictate that a shortcut menu (or context menu, as they are sometimes called) be available through the right-click of the mouse or via the "right-click" key on the keyboard. Therefore it should seem natural to use the *RightClick()* event method as the

hook to include a shortcut menu in your applications. The menu is called the same way we normally execute menus:

```
DO EditShortCut.mpr
```

You can send parameters to a menu by including an **LPARAMETERS** statement as the first line of the menu *Setup* snippet. You cannot use object references like **This** and **ThisForm** because they can only be used in object methods and menus are programs. Therefore it is common to pass an object reference to the menu:

```
DO EditShortCut.mpr WITH this
```

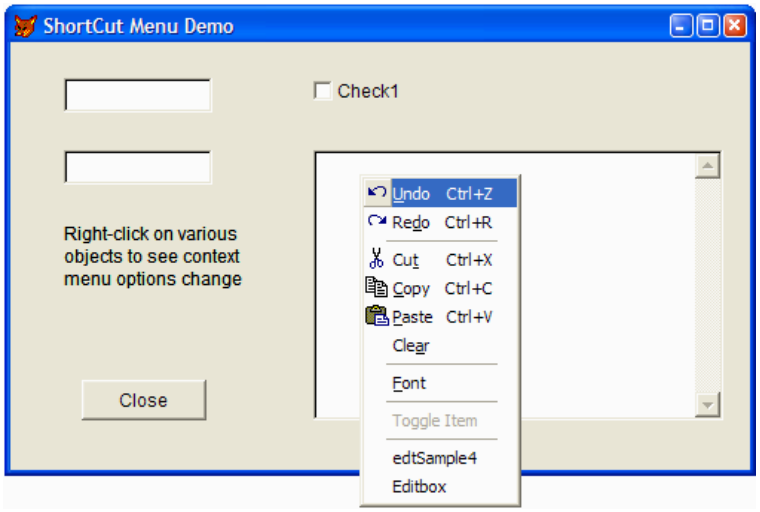


***Figure 3**. The Shortcut example demonstrates how the menu bars can be dynamically created or altered.*

The menu code is executed each time it is called. It is nothing more than a program with specific menu-based instructions. This is an important feature of the implementation since you can dynamically change the menu depending on the item that is enabled with the right-click functionality. This is the reason these menus are often referred to as context sensitive.

The example shortcut menu (see **Figure 3**) demonstrates a couple of key points in this regard. The menu is passed a reference of the object that is right-clicked. Only the form textboxes, the checkbox, and the edit box have been enabled. They all call a custom form method called *CallShortcutMenu()* and pass this method a reference to itself. The menu is called and passed this object reference. The menu has a couple of dynamic entries on the bottom. One is the *Name* of the object that called the menu, and the other is the *BaseClass* of the object. This can be important information when enabling and disabling menu bars.

The example shows how you can implement the Edit menu options so the user can access features like Cut, Copy, and Paste without the need to move to the main menu or use the keyboard. We use the information about the *BaseClass* to set the **SKIP FOR** clause for several

options. The Font menu bar is only available for the edit box, and the Toggle Item menu bar is only enabled for the checkbox.

One other tip that is demonstrated in the examples is that procedures specific to the menu are included in the Cleanup code. The Cleanup code is appended to the menu definition code generated. It has the same effect as adding procedures to the end of a program (PRG). These procedures are available in the scope of the menu program. The custom *ChangeFont()* procedure uses the object reference to use the existing font attributes when prompting the user to select the font via the **GETFONT()** function. Once the font is selected, the same reference is used to set the object's font attributes.

```
PROCEDURE ChangeFont(toCalling)

LOCAL lcFontAttributes, ;
      lcStyle

lcStyle = SPACE(0)

IF toCalling.FontBold = .T.
  lcStyle = lcStyle + "B"
ENDIF

IF toCalling.FontItalic
  lcStyle = lcStyle + "I"
ENDIF

lcFontAttributes = GETFONT(toCalling.FontName, ;
                           toCalling.FontSize, ;
                           lcStyle)

ALINES(laFontAttribute, lcFontAttributes, .T., ",")

toCalling.FontName   = laFontAttribute[1]
toCalling.FontSize   = VAL(laFontAttribute[2])
toCalling.FontBold   = "B" $ laFontAttribute[3]
toCalling.FontItalic = "I" $ laFontAttribute[3]

RETURN
```

How do I create and implement a top-level form menu? (*Example: TopLevelMain.mnx/mnt, TopLevelDemo.scx/sct*)

Top-level forms run outside of the main Visual FoxPro frame. This means that the system menu (main menu) is not available, either visibly or logically. To work around this issue, Microsoft provided menus that work in a top-level form. Top-level menus are a different animal in the Visual FoxPro kingdom.

The actual menu is created the same way you create a standard system menu. The key difference is to open up the General Options dialog and check the Top-Level Form option (see **Figure 4**). The menu generation process (GENMENU.PRG) will generate important comments and setup code needed to implement a top-level menu.

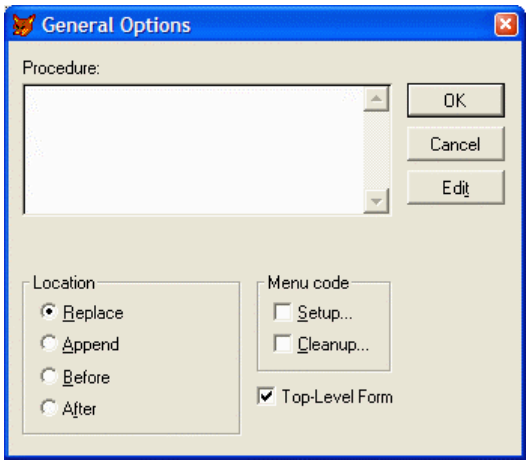


***Figure 4**. The top-level menu is set by checking the Top-Level Form on the menu General Options dialog.*

There are two keys to the actual implementation of the menu in the form. The menu is executed from the form's *Init()* method. In the simplest example:

```
DO TopLevelMain.mpr WITH This, .T.
```

The first parameter is a reference to the form. The second parameter tells the menu to rename the form's *Name* property to the name of the menu. This is helpful when releasing the menu when the form is destroyed. If the menu is not released it will remain in memory. In the form *Destroy()* method add the following code:

```
RELEASE MENU (THIS.Name) EXTENDED
```

There are different settings provided by the menu based on the parameters passed. There are 70 lines of comments generated at the beginning of the menu that sort out your options. Take a look at these if you are going to use top-level forms and menus. There are important settings to make if you have multiple instances of the forms so there is not a conflict with multiple menu definitions being in memory simultaneously.

One gotcha to look out for is that the menu takes space on the form itself. The menu literally shifts all the objects down on the form (see **Figure 5**). This can be a surprise. We recommend that you take a look at the value returned by `SYSMETRIC(20)`. This is the Windows menu height. This can vary in size depending on the user preferences set up in Windows. The example form for this section adds this value to the form *Height* in the *Init()* method. The *Top* property does not change because of the menu shifting objects downward, so it will not break code that depends on these values.

If you do try to execute a top-level menu in a form that is not a top-level form, you will be prompted with a message that tells you that you have either run the menu in the wrong form, or need to set the form *ShowWindow* property properly for a top-level form.

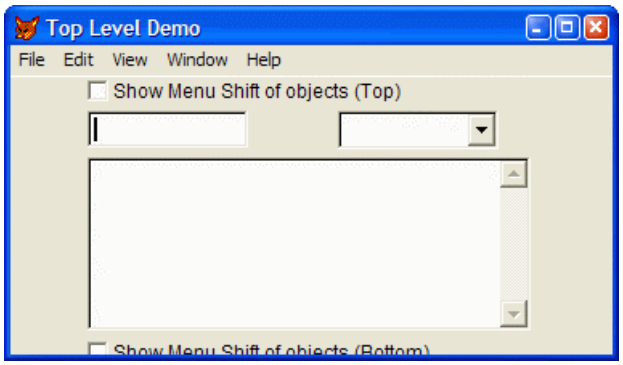


***Figure 5**. The top-level form example shows how the objects on the form are moved down by the height of the menu.*

How can I create a developer tool menu in VFP? (*Example: GeekTool.mnx/mnt*)

Visual FoxPro provides a mechanism to add menu pads directly to the Visual FoxPro development environment. We have all collected a number of tools and programs that we have purchased, developed, or downloaded that other developers have made available to enhance the development experience. One way to expose all the tools you are using is to add a menu with all your favorites.

The advantage of using a menu instead of a toolbar is that the menu does not disappear when you do a **`CLEAR ALL`**. It is common to do combination of **`RELEASE ALL`** or **`CLEAR ALL`** when Visual FoxPro gets confused after a program crashes.

Creating a developer menu is no different than developing a menu for a custom application. We recommend creating one pad and a submenu of menu bars with all the tools you want to have on the menu (see **Figure 6** for an example). On the menu General Options set the Location to Before and the pad to Window. This will place the developer pad just before the Window pad on the menu. Naturally, you can place this menu pad any where on the menu that you desire. If you do not specify the location, it will append it to the end of the Visual FoxPro menu (after the Help pad).

The example included in the chapter download is customized to the author's development environment and is provided only as an example of how to create a developer menu. You will be able to build and compile the menu, but there is a good chance that the menu will generate an error if you run it and select a menu option.

In your Visual FoxPro startup program you can establish the menu by running the following code:

```
RELEASE PAD Geeks OF _MSYSMENU
DO GeekTools.mpr
SET SYSMENU SAVE
```

The first line releases the developer menu pad if it exists. Some developers use their startup program to also clean up the environment after an application crash. If the menu pad exists and you rerun the menu program, it will create a second identical pad. The **SET SYSMENU SAVE** will tell Visual FoxPro to "memorize" the menu as the default menu.

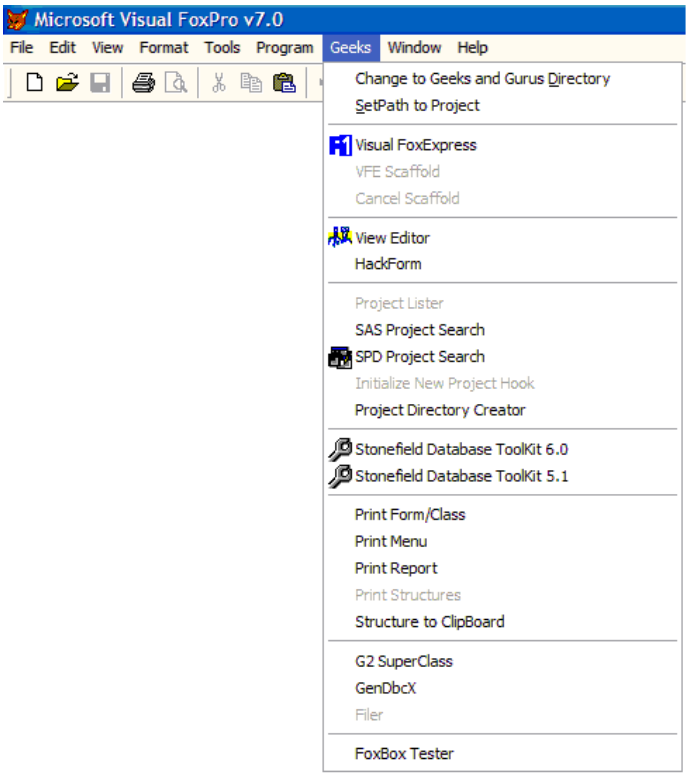


***Figure 6**. This is an example of a developer tool menu.*

If you are testing your application and it crashes and leaves the custom application menu as the system menu, you can re-establish the development menu with one line in the Visual FoxPro Command Window:

```
SET SYSMENU TO DEFAULT
```

What happens if I need to compile a VFP 7 menu in VFP 6? (*Example: MenuVFP7To6.prg*)

Many developers still need to release applications in Visual FoxPro 6, yet want the productivity advantages provided by the improvements in Visual FoxPro 7. You can develop in Visual FoxPro 7 and make the clean EXE builds in Visual FoxPro 6, provided that you do not use new or enhanced language commands, functions, or options only available in Visual

FoxPro 7. One negative side affect is that menus opened in Visual FoxPro 7 cannot be opened in Visual FoxPro 6.

The menu metadata file (MNX/MNT) format changed in Visual FoxPro 7 to accommodate the new icon capability. Two new columns were added to the menu metadata file. If you open a menu in Visual FoxPro 6 after it was opened (and automatically converted) in Visual FoxPro 7, you will get an error message: "Menu file is invalid" (see **Figure 7**).

***Figure 7**. The Visual FoxPro 6 Menu Designer displays this error message dialog when you open a menu edited in Visual FoxPro 7.*

Visual FoxPro 7 can generate MPR program code and compile Visual FoxPro 6 menus. This means if you move a project to Visual FoxPro 7 for development and do not modify the menus, you can jump back to Visual FoxPro 6 and edit, generate, and build menus with no problems.

So what happens if you accidentally (or even purposely) open a Visual FoxPro 6 menu in Visual FoxPro 7 and want to go back to the Visual FoxPro 6 menu format? You can open the MNX file as a table via the **USE** command, do a **MODIFY STRUCTURE** and manually remove the last two columns (SysRes and ResName). This is not that big of a deal, but we decided to write a program that accepts a menu name, and alters the table. Here is a partial listing of MENUVFP7TO6.PRG:

```
tcMenu  = FORCEEXT(tcMenu, "MNX")

IF FILE(tcMenu)
  lcAlias = JUSTSTEM(tcMenu)

  * Handle possible spaces in file name
  lcAlias = STRTRAN(lcAlias, SPACE(1), "_")

  USE (tcMenu) EXCLUSIVE IN 0

  * Make sure the menu is opened (not in used by another)
  IF USED(lcAlias)
    * Make sure the menu is from VFP 7 or later
    IF FCOUNT(lcAlias) > 23
      ALTER TABLE (tcMenu) DROP COLUMN SysRes
      ALTER TABLE (tcMenu) DROP COLUMN ResName
    ENDIF
  ELSE
    MESSAGEBOX("Could not open the menu, please try again.", ;
               0 + 16, _screen.Caption)
```

```
    llReturnVal = .F.
  ENDIF

  USE IN (SELECT(lcAlias))
ENDIF
```

If you do convert a menu back to Visual FoxPro 6 and reopen it in Visual FoxPro 7, the icon information previously added will be deleted and you will need to reassign icon resources to the menu options.

Both fixes described in this section will create a BAK file (of the MNX) and a TBK file (of the MNT). So if you did not intend on converting the file back to the Visual FoxPro 6 format or needed to recover the icon/resource information, you still have one last chance to recover it.

How can I fix the disabled menu after a report preview?

There is a known bug in Visual FoxPro with the report preview window causing a menu to be disabled after the report preview is exited. The menus can be disabled if the user either resizes the report preview window, maximizes or minimizes it, or closes it with the close button (X in the upper right corner).

The simple work around is to wrap the **REPORT FORM** code with a **PUSH MENU** and **POP MENU**.

```
PUSH MENU _msysmenu
REPORT FORM MyReport PREVIEW
POP MENU _msysmenu
```

The other workaround is to execute the menu program again after the report preview is closed. The PUSH and POP menu will consume memory during the report execution, but our experience is that it is not something to worry about.

A partial replacement for the Menu Designer (*Example: MenuDesigner.pjx/exe*)

The Visual FoxPro Menu Designer is fundamentally the same old Menu Designer that we have been working with for years, since the days of FoxPro 2.6. There are a number of issues that have not been addressed that have pushed us over the edge to create the first attempt at replacing the Menu Designer. The result of this endeavor is something we are calling the G2 Hack Menu form.

There are a number of issues that the tool attempts to address (see **Figure 8**). The first is the woefully small area a developer is given to enter in the menu prompt and the results or action of the menu option. The text boxes provided on the Menu Designer are way too small. The other major frustration addressed is the need to open up a modal form for the Options. Jumping in and out of the Options dialog is a painful experience when you are attempting to view all the options to make sure they are correct before a build. The fact that the Menu Designer only shows 10 menu bars to start is limiting when the designer is not resizable.

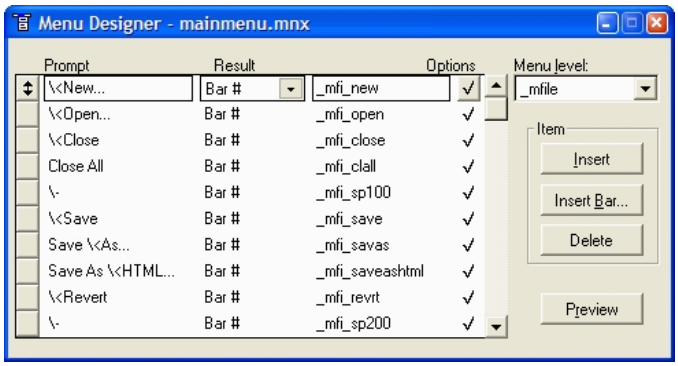


***Figure 8**. The Visual FoxPro Menu Designer is showing its age in a state-of-the-art development environment.*

Another thing that makes the Visual FoxPro Menu Designer difficult to use is the way you navigate from one level to another. The Menu Level combo box is not exactly a friendly interface. The last thing we dislike (like there have not been enough already) is that the menu options for the Menu Designer are scattered across two menu pads. The Menu pad is obvious enough to gain access to the Quick Menu, the adding and deleting of menu bars, the menu preview as well as the MPR generation process. You also have to pay attention to the View pad to gain access to the menu's General Options and the Menu Options.

One word of caution when working with this tool and other Visual FoxPro metadata tools: Make sure you make a backup of your metadata before hacking it. If you change something in the metadata and that change is not supported by the Menu Designer or the GENMENU.PRG, you can disable the menu and the ability to edit it in the native tools. This is the source code to your applications; safeguard it before hacking into it. The authors take no responsibility for your hacking actions.

The intent of the G2 Hack Menu is not to generate a new Menu Designer from the ground up. The original specifications dictate that it is 100% compatible with the native Visual FoxPro metadata. The reason for this is that Visual FoxPro developers still want to be able to use the native Visual FoxPro tools, and as noted later in this section, the tool does not completely replace all the functionality of the native designer. It is capable of editing regular menus, shortcut menus, and top-level form menus.

The first benefit of the G2 Hack Menu is the ability to see and edit all the menu bar information on one page (see **Figure 9**). The Fundamentals page shows menu bar information for each menu item. If you look at the menu metadata file (MNX), you will see that the Fundamentals page shows records from record 3 to the end of the table. Also note that the TreeView shows menu pad and bar items. There are multiple records in the menu metadata for menu pad items, so as you navigate through the records there will be records on the Fundamentals page that are not reflected in the TreeView. The general page exposes records 1 and 2 of the menu metadata. The primary focus of this page is to expose the Setup and Cleanup code as well as the procedure code. Toggling between the Fundamentals and General pages will restrict the records that are in scope since they address different aspects of the menu metadata. The current record number of the metadata displayed is shown at the bottom of the form.

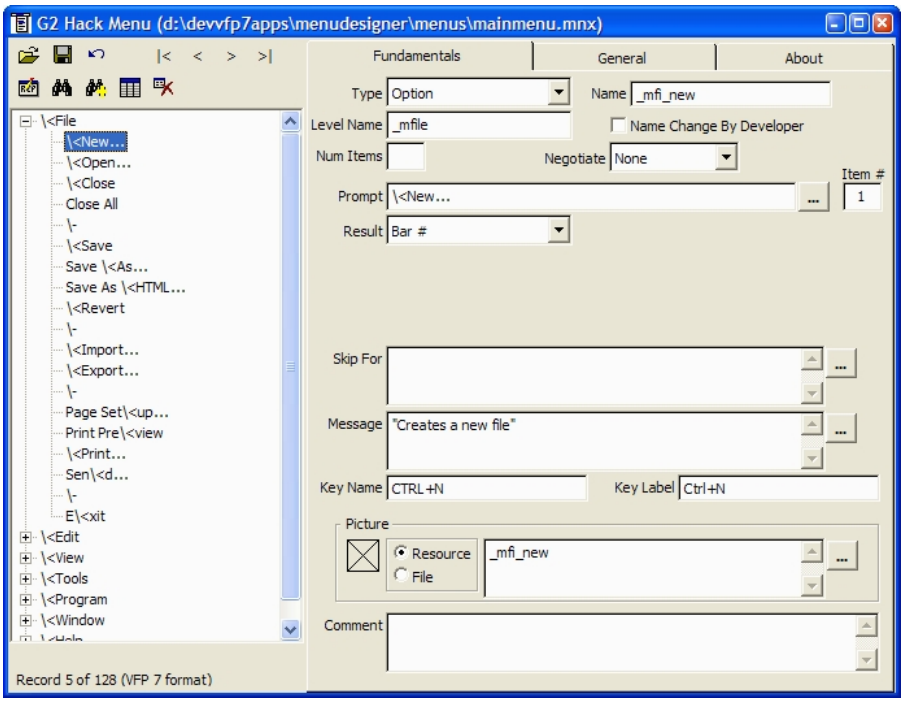


***Figure 9**. The G2 Hack Menu tool is the first step at creating a replacement for the Visual FoxPro Menu Designer.*

To address the restrictive navigation of the Menu Designer, the G2 Hack Menu provides both a TreeView and navigation buttons. The TreeView allows you to quickly drill down the menu tree just like you would cascade the menu to pick the option you are accessing. The navigation buttons provide the ability to traverse the records in the menu metadata. As you move to a new record, the TreeView is expanded to expose the record you are on in the metadata.

If you want to look up a specific record, you can use the search (binoculars) and search again (binoculars with plus sign) buttons on the form toolbar. The search feature will present a dialog (see **Figure 10**) that allows you to select the column of the metadata to search in and a textbox for the text to search. If a record is found, it is displayed in the form; otherwise, a message is displayed indicating the failure to find the text. One caution when searching the prompt field is to make sure that you include the hot key indicator in the text that you are searching. The text comparison used in the search is made using the $ operator and is case-insensitive. This means that searching the prompt column for "\<View" will locate "Pre\<view" and "\<View" menu bars. This feature uses the LOCATE (search) and CONTINUE (search again) commands so each time you use the search functionality it will find the first occurrence of the text.

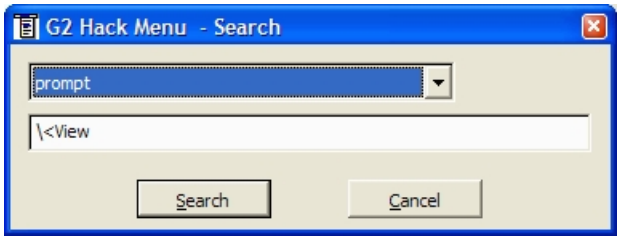


***Figure 10**. The G2 Hack Menu search dialog lets you pick the column to search and the text to be searched.*

The command window option provides the developer with the ultimate hack tool. It executes a single command (future implementations will allow full programs to be run). If you are used to doing mass replacements of **SKIP FOR** conditions via a **REPLACE ALL** command, you can perform these within the tool and with the advantage of being able to revert to the original settings by not saving the changes.

This brings up the next feature, which is that the changes are fully revertible because the tool implements buffering of the metadata. The changes are not saved until you press the Save button on the toolbar, or close the form and respond Yes to the question about saving your changes.

If you are comfortable with the technique of browsing the metadata, but like the user interface of the G2 Hack Menu, yet find a limitation of the interface, you can still browse the metadata within the tool. Again, this technique is safer than just browsing the metadata because it is buffered and you can reverse your changes.

The Visual FoxPro Menu Designer does a darn good job of keeping logically deleted records from hanging around. We still provide a **PACK** command just in case you find memo or record bloat in the metadata.

Finally, the G2 Hack Menu supports both Visual FoxPro 7 and menus built in previous versions of Visual FoxPro. The metadata layout changed with Visual FoxPro 7 to accommodate the new icon feature on the menu. If you are editing a menu created and edited prior to Visual FoxPro 7, the icon functionality is made invisible.

There are a couple of items on the user interface that expose information internal to the Menu Designer that we chose to display, but left read-only to protect developers from easily corrupting the way the menus are generated. Please note that this does not mean you will not be able to change other metadata to the point that it is corrupt, just that we wanted to protect the obvious ones. The objects protected include the Type combo box (General and Fundamental page) and the Name Changed by Developer checkbox on the General page.

There are a few features in the Visual FoxPro Menu Designer that have not made it into this cut of the G2 Hack Menu tool. The first is that you cannot add/delete menu pads or bars. The tool does not provide a callout to the MPR generation, nor is does it have a menu preview mode, although the TreeView provides the basic visual representation. It has no mechanism to add Visual FoxPro bar resources either. We want to add some other search capabilities like

string search in all fields and exact searches, have the ability to find the next changed item, and have the changed record icon display as soon as a change is made. Another feature on the enhancement list is to provide a mechanism to open up the various code edit boxes into the program editor to show code colorization and more importantly, allow the power of IntelliSense to be available to the developer. One other enhancement on the list is to make the tool resizable. Check the various Web sites on the About page for future updates of this evolving developer tool.

Coverage Profiler

The Coverage Profiler was introduced in Visual FoxPro 6. Recording coverage logs through the Visual FoxPro debugger was introduced in Visual FoxPro 5. The Coverage Profiler is an excellent way to determine where the performance bottlenecks are in your applications (Profile mode) as well as viewing which code was actually executed in the testing you performed on the application (Coverage mode).

How do I start recording coverage logs?

The Coverage Profiler is pointless unless you have a coverage log for it to analyze. The log is created by the debugger recording statistics on each line of code it executes. This logging can be turned on two ways.

The first way to turn on the coverage logging is in code. Determine the place that you want the logging to begin. At that point in the code add the following line of code:

```
SET COVERAGE TO c:\temp\billingperformance.txt
```

This turns the coverage logging on as well as directs the statistics to a file with the specified name. If the file did not exist, it is created. If it exists, it will be overwritten unless you use the **ADDITIVE** clause after the file name. At the point in the code that you want to shut off the collection of statistics enter in the following line:

```
SET COVERAGE TO
```

This turns the coverage logging off as well as closes the coverage log file. We have noticed in some versions of Visual FoxPro that the coverage log file was not always closed until a **CLOSE ALL** was executed. This is definitely fixed in Visual FoxPro 7.

The other way to start coverage logging is to use the debugger user interface. If you are using the Debug Frame you can toggle coverage logging from the Tools | Coverage Logging menu. If you use the FoxPro Frame for debugging, you only have the option of clicking the Toggle Coverage Logging toolbar button. Either way, when you toggle it on you are presented with the Coverage dialog (see **Figure 11**). You enter in the file name (full path unless you want it created in the default Visual FoxPro directory) and determine whether you want a fresh file or to append on to an existing log.

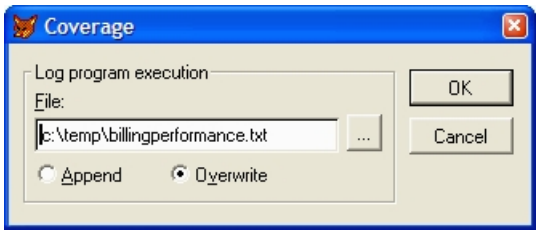


***Figure 11**. The Coverage dialog is presented if you toggle coverage logging on via the debugger interface.*

To turn off the logging interactively you select the same menu option (Debug Frame) or press the Toggle Coverage Logging toolbar button (Debug or FoxPro Frame). After you toggle the coverage logging off, you can open up the text file with the Visual FoxPro editor or use the Coverage Profiler to do some sophisticated analysis.

You must have both the coverage log and the source code that was executed to use the Coverage Profiler. It needs the source code to mark up which lines of code were not executed and to show the performance timings for the lines that were execute. If someone sends you a coverage log, you will only be able to open it with a text editor if you do not have the exact source code that was run.

One observation of interest is that if you have the Visual FoxPro debugger active, IntelliSense turned on, and type in the Command Window or an editor, you will see code executed in the debugger as IntelliSense processes. The coverage logs never show any statistics for the IntelliSense engine.

What are the different columns in the coverage log files? (*Example: MenuDesignerCovLog.txt*)

The coverage log is a text file that is generated by Visual FoxPro if you have turned coverage on while running code in your application. This text file is a comma-separated file with six columns of information to assist you in finding performance bottlenecks and to determine which code was executed and not executed.

The first column is the execution time for the line of code. The time is either the execution time for the line of code, or if the line calls other procedures/functions/methods it is the time it takes for all subordinate code to execute. The time is measured in seconds, accurate to six decimal places.

The second column is the name of the object containing the code that was executed. For example, if the code is in a form, the name of the form is recorded. The column is left blank if the code is in a procedure or program.

The third column is the name of the method, procedure, or function being executed. If the code is executed in a method, the name of the object is attached to the method name in the object.method format.

The fourth column is the line number of the code that was executed. The line number is the actual line number from the start of the program or method. If a line of code is broken up into several lines with continuation character (semicolons), it will be the last line of that code. This line number can be used to open up the editor with `EDITSOURCE()`.

The fifth column is the fully qualified file name of the object containing the code that executed.

The last column is the calling stack level for the executing code. We could not find any part of the Visual FoxPro Coverage Profiler that does anything with this information, but it is available if you want to write an add-in that would use it to analyze the code. One idea of an add-in that would use the calling stack is to see what the deepest level of calls is in the code that was tracked in a coverage log.

How do I register a Coverage Profiler add-in?

The Coverage Profiler is extendible via the add-in capability. Add-ins can be any Visual FoxPro executable code such as a program, class, application, form, or EXE. You start the registration process by pressing the Add-ins button on the Coverage Profiler toolbar (fourth button from the left). This brings up the Coverage Profiler Add-ins dialog. Type in the fully pathed module or select the module that you want executed via the Open File dialog (ellipsis button next to the Add-in textbox).

The add-in must be run every time you want to use it with the Coverage Profiler. There is no way to have the extension run when the native Coverage Profiler is started. You can subclass the native Coverage Profiler to run add-ins on startup if you like.

The "Register this Add-in after running" checkbox is important to check if you want quick access to this add-in the next time the Coverage Profiler is run. Once registered, the add-in appears in the dropdown the next time you run the Coverage Profiler.

Where are Coverage Profiler preferences and add-in registrations saved?

The Coverage Profiler keeps preferences and add-in registration in the Windows Registry (see **Figure 12**). Both Visual FoxPro 6 and 7 track these preferences independent of each other.

Visual FoxPro 7 tracks many more options (30 items) than Visual FoxPro 6 (14 items). All the Registry entries are tracked in the following Registry key:

```
HKEY_CURRENT_USER\Software\Microsoft\VisualFoxPro\7.0\Coverage
```

Options saved in the Registry include Font attributes (size, name, bold, italic), whether the Coverage Profiler runs in a top-level form or within the Visual FoxPro frame (similar to the debugger option), frame attributes (height, width, top, and left), main dialog attributes, zoom form attributes, mark attributes (execute and non-execute markings, and whether all marked), profile mode indicator, smart pathing, stack XML extended tree, and the zoom mode.

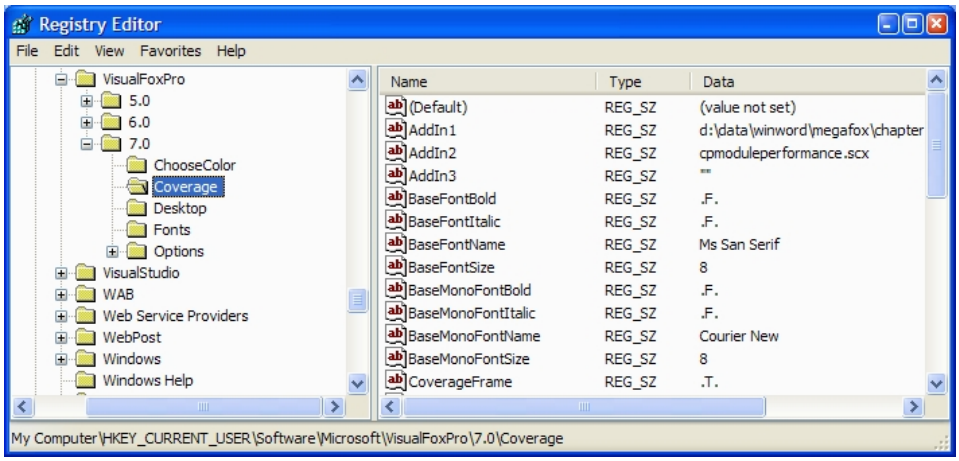


***Figure 12**. The Coverage Profiler preferences are stored in the Windows Registry.*

How can I delete Coverage Profiler add-ins I no longer want registered?

The Coverage Profiler add-ins are registered and stored in the Windows Registry. Add-ins are easy to register via the Coverage Profiler Add-ins dialog, but there is nothing in the Coverage Profiler tool that allows you to remove the add-in from the list of registered add-ins (see **Figure 13**).

Each add-in is a key value under the Visual FoxPro Coverage Registry key. They are named AddIn1, AddIn2, and so on (see Figure12). The only way to remove the add-in is to edit the Registry and delete the keys.

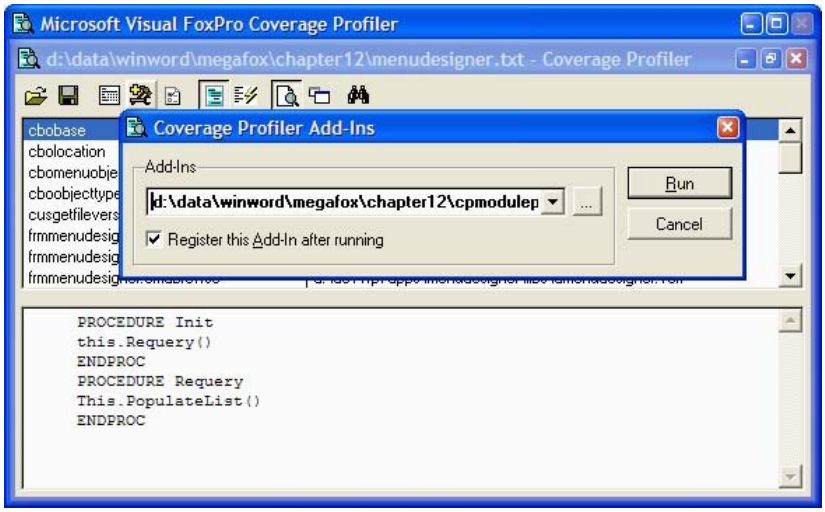


***Figure 13**. The Coverage Profiler has a mechanism to register add-ins, but no way to natively unregister the add-ins.*

Coverage Profiler add-in to summarize module performance

(*Example: CpModulePerformance.scx, CpModulePerformanceReg.prg*)

The standard profiling mode of the Coverage Profiler shows the performance of the individual lines of code that were executed. What if you wanted to see the combined performance numbers of all the lines of code within a method or for a module? One solution would be to pull out a calculator and add up all the lines displayed in the Coverage Profiler. The flaw with this is that the Profiler mode shows the average execution, not the actual execution speed. The more appropriate method would be to create an add-in to the Coverage Profiler that summarizes the line execution details into a nice form for the developer. This section will discuss the program that does this and also demonstrate how to register a Coverage Profiler add-in that calls a form that is displayed when the add-in is activated.

The CPMODULEPERFORMANCEREG.PRG (see **Listing 2**) is a program that registers the add-in and adds a toolbar button to the main Coverage Profiler form. The Coverage Profiler passes a reference of itself to the add-in; therefore you must accept a parameter in the add-in program. The majority of the code in the program is safety code. What we mean by this is that it checks to make sure it is called from the Coverage Profiler, it has not been called previously in this Coverage Profiler session, and ensures that the toolbar button is instanced only once.

***Listing 2**. Code from CpModulePerformanceReg.prg is an example of how to write code for a Coverage Profiler add-in.*

```
LPARAMETERS toCoverage

LOCAL llReturnVal, ;
      loControl

IF VARTYPE(toCoverage) # "O" OR TYPE("toCoverage.cAppName") # "C"
   MESSAGEBOX("You need to be running the VFP Coverage Profiler " + ;
              "for this program to be effective.", ;
              0 + 16, ;
              _screen.Caption)
   llReturnVal = .F.
ELSE
   llReturnVal = .T.

   * Loop through all Coverage profiler toolbar controls to see if the
   * cmdModPerformanceButton is already instantiated. We do not want
   * more than once instance of this control registered.

   FOR EACH loControl IN toCoverage.frmMainDialog.cntTools.Controls
     IF LOWER(loControl.Class) == "cmdmodperformancebutton"
        WAIT WINDOW "Module Performance Button already loaded!" NOWAIT
        llReturnVal = .F.
        EXIT
     ENDIF
   ENDFOR

   IF llReturnVal
      * Button is not on Coverage Profiler, so we add it.
      toCoverage.frmMainDialog.AddTool("cmdModPerformanceButton")
   ENDIF
ENDIF
```

```
RETURN llReturnVal


DEFINE CLASS cmdModPerformanceButton AS cmdCoverageToolButton
*  This button subclass is of the CoverageToolButton Class
*  (see below)
   Caption     = "MP"
   ToolTipText = "Module Performance Analyzer Add-in"
   AutoSize    = .F.
   Width       = 22
   Height      = 23

   PROCEDURE Init
      IF VERSION(5) > 600
         this.SpecialEffect = 2
      ENDIF
   ENDPROC

   PROCEDURE Click
      thisformset.RunAddIn('CpModulePerformance.scx')
   ENDPROC

ENDDEFINE


DEFINE CLASS cmdCoverageToolButton AS CommandButton
* This base class is borrowed directly from Lisa Slater Nichols.
* It integrates the button into the toolbar in an appropriate fashion.
* This class also includes basic error handling as built into the
* Cov_standard class.

   lError   = .F.
   AutoSize = .T.                      && Text will fit automatically

   PROCEDURE Init
   * Use some formset properties to make the new tool "fit in"

      WITH thisformset
         this.FontName   = .cBaseFontName
         this.FontItalic = .lBaseFontItalic
         this.FontBold   = .lBaseFontBold
         this.FontSize   = .nBaseFontSize
      ENDWITH

      * Now use the container's physical properties
      * to fit in there as well:
      THIS.Autosize = .F.

      WITH THISFORMSET.frmMainDialog.cntTools
         THIS.Top = .Controls(1).Top
         THIS.Height = .Controls(1).Height
      ENDWITH

   RETURN (NOT THIS.lError)

   PROCEDURE Error(tnError, tcMethod, tnLine)
   * Designed to use the FormSet's error method which, in this
   * case does nothing more than put up an error MessageBox.
```

```
      THIS.lError = .T.

      IF TYPE("thisformset.BaseClass") = "C"
         thisformset.Error(tnError, this.Name + ":" + tcMethod, tnLine)
      ELSE
         ERROR tnError
      ENDIF

   ENDPROC
ENDDEFINE
```

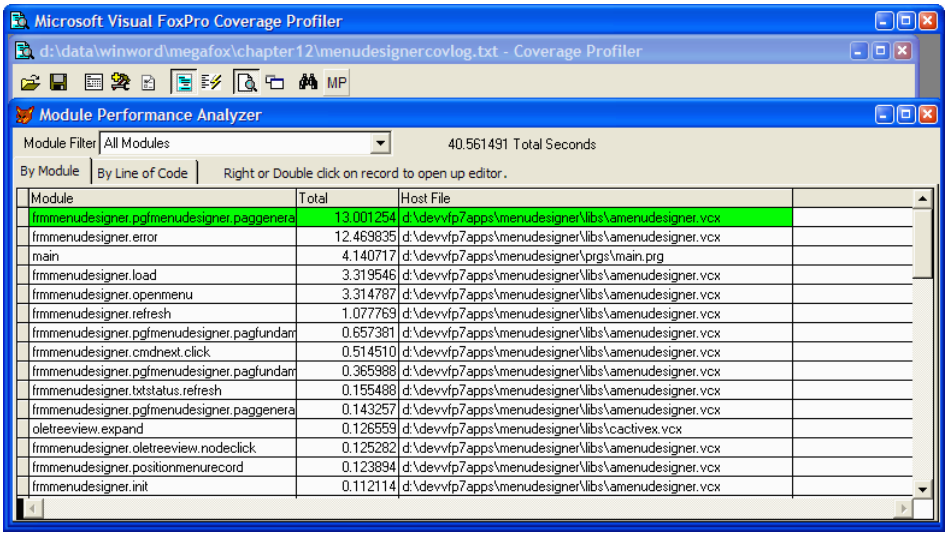


***Figure 14**. The Coverage Profiler Module Performance Analyzer shows the total time it takes for each method/module and the execution time for each line (on the By Line of Code page).*

The form starts out by performing three queries on the FromLog cursor (created by the Coverage Profiler). The FromLog cursor contains the timings on each line of code that is executed and tracked in the coverage log by the debugger. The first query is a summary of the code execution by each module. This cursor is used on the By Module page. This cursor is sorted by total time for the module in descending order. This is done to help the developer determine the slowest modules in the analysis. The second query is for all lines and the time it takes for each line for each execution. This cursor is sorted by Module, by line number, and by execution time. This cursor is used by the By Line of Code page. Developers can use this information to see which lines are taking the longest and whether there is a significant difference each time the line of code is executed (see **Figure 14**). It is not uncommon to see the same line of code degrade in performance when it is executed over and over, especially if there is a memory leak in the method. The last query is used to populate the module combo box.

The only real important property for the add-in form is that it must have the *ShowWindow* property set to 1 – In Top-Level Form because the Coverage Profiler is a Top-Level formset

based tool. The total number of seconds will reflect the number of seconds for the modules included in the list. This can be filtered by selecting a specific module using the combo box. When you select a specific module, a filter is applied to the two performance analysis cursors.

The two grids have *DblClick()* and *RightClick()* calls to open up the source code in the appropriate editor using the new **EDITSOURCE()** function, which is the only reason this tool requires Visual FoxPro 7. If you have not yet upgraded to Visual FoxPro 7 and want to use this tool, we suggest that you override the form's *EditModule()* method. Activating the source code editor allows the developer to make changes live or at least view the exact code that might be a potential bottleneck.

Class Browser

The Class Browser is a powerful tool that helps Visual FoxPro developers manage classes and class libraries. From the Class Browser you can modify, delete, and subclass a class. You can redefine the class superclass, and change the icon that is displayed in the Project Manager and Class Browser. Gaining a full grasp of the capabilities of the Class Browser can reap important benefits for a Visual FoxPro developer.

How can I set the default file to be opened when Class Browser is started?

Ever have one of those days where you work on refactoring a class or developing a set of classes in the same class library all day? You keep opening up the Class Browser and picking the same class library over and over. Ever wish that you could set it up so that when you open up the Class Browser it opens up a specific class library? Well, you can if you set it up to do so.

First open up the Class Browser and open up the class library that you want to be the default class library opened when it starts. In the Command Window:

```
_oBrowser.SetDefaultFile()
```

If you want to clear the default class library execute the following statement:

```
_oBrowser.ResetDefaultFile()
```

How do I open the Class Browser with a specific class?

When starting up the Class Browser programmatically, you can open to a specific class and even preselect the class and a property or method of the class.

The first parameter passed to the Class Browser is the class library. The class library must be on the Visual FoxPro path or be fully qualified to have the class library open successfully. The second parameter is the class name, followed by a period and then the property or method. If you just pass the class name as the second parameter, the class library will be open, but no class is selected. You need to pass a property or method attached to the class name to get the class highlighted in the left pane (TreeView). Here is an example call to the Class Browser to open up with the lCopyAppToTestDirectory property and the phkDevelopment class selected in the CPhkBase2 class library.

```
DO (_browser) WITH "CPhkBase2"," phkDevelopment.lCopyAppToTestDirectory"
```

This can be handy if you are trying to save time opening up the same class library over and over during a development session. Another shortcut is to start the Class Browser, right-click on the Open command button, and select a class library from the most recently used list that drops down.

How can I move and copy classes between class libraries?

Using two instances of the Class Browser allows developers to move and copy classes between class libraries. The key to moving or copying classes is to drag the class from the class icon in the top left corner of the Class Browser. It is important to hold the Ctrl key down first before selecting the icon. Drag-and-drop between class browser instances defaults to moving the class. To copy the class, hold the Ctrl key down during the drag to the second instance. You can drop the class on any part of the second Class Browser.

How do I rename methods and properties without opening the class?

You can rename methods and properties by right-clicking on the property or method and selecting Rename... on the shortcut menu. A dialog is presented that allows you to rename the property or method (see **Figure 15**). Please remember that renaming a method or property does not magically rename the references in the method code. You will need to manually search and replace any references that are changed (there is one caveat discussed later when manually replacing references is unnecessary).

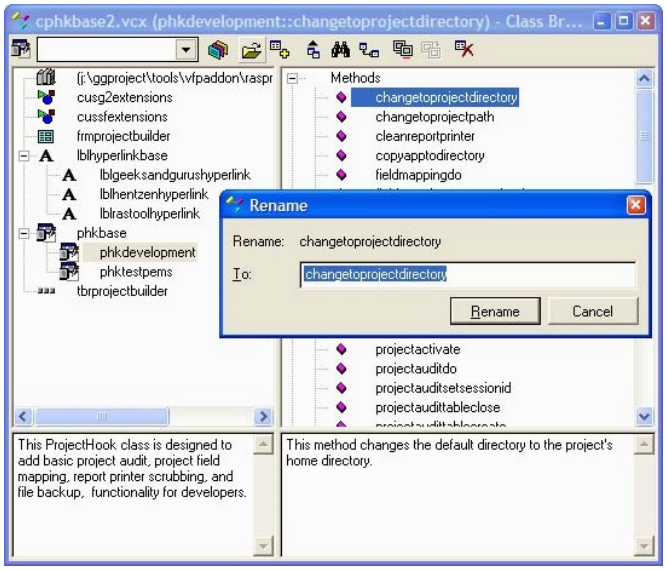


***Figure 15**. The Class Browser provides a method and property renaming feature without opening up the class.*

How can I safely change a class name without breaking references to subclasses?

It never seems to fail. You start out developing this cool class and you have it in a class library that at 3:00 in the morning seemed to make sense. You develop it, you tweak it, and after significant testing you implement the class in a number of forms and other classes in a project. The next morning, after getting some needed sleep, and after that first can of Coke you realize that a different class library is the more appropriate location for the class. Moving or renaming a class or a class library can cause all kinds of trouble for Visual FoxPro developers. If you rename a class, or move a class, all the other classes and forms that contain this class will ask you to locate the class each time you open them in the designers. This can be a big pain, especially if the classes being renamed are an integral part of a framework. So how can we avoid such pain?

The Visual FoxPro Class Browser will adjust the name and/or location of all class library and form files in the Class Browser automatically if a subclass or instance exists for the class changed. You can also open Visual FoxPro project files, which loads in all class library and form files for the project. Therefore, if you rename a class it is recommended that all class library and form files that use or are a subclass of the changed class be loaded in a Class Browser window to have the reference automatically updated.

One thing that it will not automatically change is code that references the class like:

```
thisform.oRegistry = NEWOBJECT("cusRegistry", "CFramework")
this.oBusiness     = CREATEOBJECT("cusInvoiceBO")
```

The Class Browser is only smart enough to fix the object inheritance hierarchy, not references to the classes in code. This is something you will need to handle manually. But it is much better than handling the code and the objects that are broken because the name of the class was changed. If the class is from a common library and used across several projects, you want to be sure to open up each of the projects affected, or reconsider your decision to rename the class.

How can I test classes from the Class Browser?

The Visual FoxPro Class Browser has some terrific drag-and-drop capabilities that assist Visual FoxPro developers in testing classes. The various scenarios will allow developers to instance the classes on the Visual FoxPro desktop, instance the class on a live object, instance the class in the Form or Class Designer, or create instance code in the Command Window.

All drag operations from the Class Browser are started by selecting the class in the TreeView and then dragging the class icon located in the upper left corner.

Developers can instance a class on the Visual FoxPro desktop by dragging-and-dropping the class icon onto the Visual FoxPro desktop. If you hold the Shift key during the drag-and-drop the class is created, but it is not visible. Holding the Ctrl key as you drag-and-drop the class will also display any errors that occur while the object is being created (see **Figure 16**). These errors are normally ignored when creating classes in this fashion. Classes are instanced

as objects on the Visual FoxPro **_screen** object. The object references will be the class name with a numeric counter attached. Strangely, they cannot be seen in the Property Sheet, but can be seen in the debugger Watch window by adding **_screen** and are available to IntelliSense. If we dragged a class called txtBase to the Visual FoxPro desktop, we could reference it in the Command Window using code like this:

```
?_screen.txtBase1.Name
_screen.txtBase1.Visible = .F.
_screen.RemoveObject("txtBase1")
```

This allows you to set properties and call methods to test the class live on the desktop.

***Figure 16**. The Class Browser displays error information when holding the Ctrl key and dragging the class to the Visual FoxPro desktop.*

If you drag-and-drop the class icon onto the Command Window, Visual FoxPro will generate the **NEWOBJECT()** function call in the Command Window and instance the class:

```
oTxtbase1 = NEWOBJECT("txtbase","d:\data\winword\megafox\chapter12\ch12.vcx")
```

Subsequent instances of the same class are named the same with an incrementing numeric value attached to the end of the name. The class instances are directly available as memory variables and can be seen as instances in the Class Browser in the right side pane.

You can drag-and-drop classes onto a running form, any class that is a container, or to a form or class opened up in the designer in the same manner as described for the desktop or Command Window. To add an instance of the selected class to a live form programmatically, you select the class in the Class Browser that you want added to the form. Put the mouse on the form where you want the class added and execute the following code in the Command Window:

```
_oBrowser.FormAddObject(SYS(1270))
```

Now you can test the class instanced on the form. If you are satisfied that it is working and you have the form instance with a variable called oFrmTest1, you can then save the live form with code similar to this:

```
oFrmTest1.SaveAs("MyCustom")   && Direct save to SCX
oFrmTest1.SaveAsClass("MyClassLib.vcx", "frmTest2")   && Subclass of the class
```

A gotcha here is that the form or class cannot be saved to the file of the form or class currently instanced since it is already an object in memory. Therefore, you will have to save it to a different form or class and work later to rename it.

How can I view and edit superclass code via the Class Browser?

Most Visual FoxPro developers are familiar with a free tool called SuperClass. The SuperClass tool can do a number of things, but the feature it is most famous for is opening up the superclass method code of the method you are editing and allowing you to directly edit it without opening up the superclass first. This powerful tool was written by Ken Levy (who also happens to be the architect and developer of the Class Browser back in the day when he was a contractor doing development for Microsoft on the Fox Team). This same feature is available if you are editing a class via the Class Browser.

***Figure 17**. The Edit ParentClass Method toolbar button is available if you edit a class from the Class Browser.*

The Edit ParentClass Method button appears on the Visual FoxPro toolbar whenever a Class Browser window is active (see **Figure 17**). This button allows you to view and edit the immediate parent class method while you're in the Form or Class Designer method editor. If you are not interested in this functionality, you can disable this by setting a property of the Class Browser using the following code in the Command Window when the Class Browser is open:

```
_oBrowser.lParentClassBrowser = .F.
```

You will be prompted to save the superclass code after you close the window. The only disadvantage to editing code in this manner is that you lose editor colorization and the IntelliSense is not active.

Does the Class Browser add-in retain the Regional Settings for time and date? (*Example: CbChangeDateFormat.prg*)

We like when applications respect the Windows Regional Settings for date and times. We write our applications to respect the user selections via the `SET SYSFORMATS ON` at the beginning of the applications. We noticed that the Class Browser does not respect the Regional Settings and defaults to `SET CENTURY OFF`. To combat this situation we wrote a simple add-in to the Class Browser that sets the date and time settings to the developer preference (see **Listing 3**).

This Class Browser add-in can be run two ways. The first way is to just run the program from the Command Window. The add-in will recognize the fact that it is not being run from the Class Browser and will toggle into "registration mode." Registration mode adds a record to the BROWSER.DBF file. This table contains the registration information of all the add-ins. The Browser table is self-explanatory and documented in the Visual FoxPro Help file. The key points we want to make about the registration of this add-in is the Method column is "Activate" and the Program column is the name of the program being run. By setting the Method column to "Activate," we are telling the Class Browser to execute this add-in every time the Class Browser is activated. The registration code only needs to be run once on each development copy of Visual FoxPro you have loaded. If it is run more than once it will update the registration record.

***Listing 3**. Class Browser add-in that changes the date and time displayed in the Class Browser to the ones set up in the Windows Regional Settings.*

```
LPARAMETERS toBrowser

LOCAL lcName                          && Name of the Add-in
LOCAL lcComment                       && Comment for the Add-in
LOCAL lnOldSelect                     && Save for reset later

* Self registration if not called form the Class Browser
IF TYPE("toBrowser") = "L"
  lcName    = "Rick Schummer's Date Setting Changer"
  lcComment = "Developed by RAS for online forum discussion and example"

  IF TYPE("_oBrowser")= "O"
    * If Class Browser is running, use Addin() method
    _oBrowser.Addin(lcName, STRTRAN(SYS(16),".FXP",".PRG"), ;
                    "ACTIVATE", , , lcComment)
  ELSE
    * Use the low level access of the Browser registration table
    IF FILE(HOME() + "BROWSER.DBF")
      lnOldSelect = SELECT()

      USE (HOME() + "BROWSER") IN 0 AGAIN SHARED ALIAS curRASDateChanger
      SELECT curRASDateChanger
      LOCATE FOR Type = "ADDIN" AND Name = lcName

      IF EOF()
        APPEND BLANK
      ENDIF

      * Always replace with the latest information
      REPLACE Platform WITH "WINDOWS", ;
              Type     WITH "ADDIN", ;
              Id       WITH "METHOD", ;
              Name     WITH lcName, ;
              Method   WITH "ACTIVATE", ;
              Program  WITH LOWER(STRTRAN(SYS(16), ".FXP", ".PRG")), ;
              Comment  WITH lcComment
      USE
```

```
      SELECT (lnOldSelect)
    ELSE
      MESSAGEBOX("Could not find the table " + HOME() + "BROWSER.DBF" + ", ;
                 please make sure it exists.", ;
                 0 + 48, ;
                 _screen.Caption)
    ENDIF
  ENDIF

  RETURN
ELSE
  * Check to see if we really got called from the Class Browser
  IF !PEMSTATUS(toBrowser, "lFileMode", 5)
    RETURN .F.
  ENDIF

  * Now the simple stuff to change the Date environment
  * setting which is specific to the private datasession.
  * This setting is driven from the developer's Windows'
  * Regional Settings.
  SET SYSFORMATS ON

  * Then refresh the Class Browser to reflect the change
  toBrowser.Refresh()
ENDIF

RETURN
```

If the code is executed by the Class Browser (remember that the Class Brower will automatically execute this code each time it is activated), it will have a reference to the Class Browser that ran this code. The Class Browser always passes a reference to itself when it executes an add-in. The code determines whether the reference really is a Class Browser. If it is, the `SET SYSFORMATS ON` is executed and the instance of the Class Browser is refreshed. The date and time at this point should be in the same format as the Windows Regional Settings.

How do I create a Class Browser add-in to set the font to my favorite? (*Example: CbChangeFont.prg*)

Almost every aspect of the Visual FoxPro development environment has a way to change the font to the font of your choice. The Class Browser is no different. Each time the Class Browser is started you can right-click near the toolbar at the top and select the Font… menu option (see **Figure 18**). Sound like fun? We did not think so and wrote a quick add-in to change the font each time the Class Browser is started (see **Listing 4**).

Just like the add-in to change the format of the date and time to the Windows Regional Settings, this Class Browser add-in has two modes, a "registration mode" and the action mode. See the section "Does the Class Browser add-in retain the Regional Settings for time and date?" for a discussion on the "registration mode."

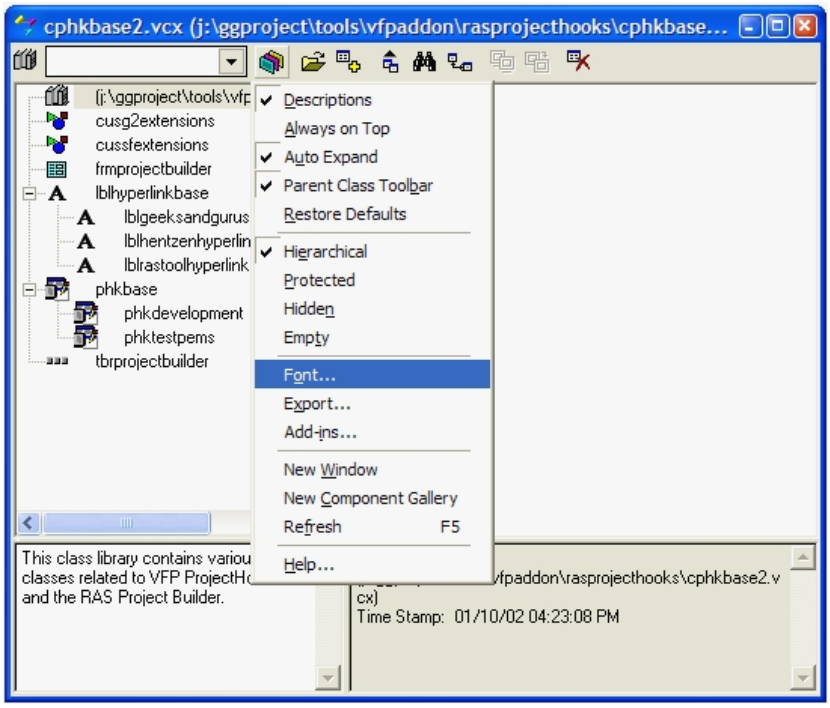


***Figure 18**. You can use the Class Browser shortcut menu to change the font each time you start up the Class Browser.*

***Listing 4**. Class Browser add-in that changes the font of the Class Browser each time it is started.*

```
LPARAMETERS toBrowser

LOCAL lcName                            && Name of the Add-in
LOCAL lcComment                         && Comment for the Add-in
LOCAL lnOldSelect                       && Save for reset later

* Self registration if not called form the Class Browser
IF TYPE("toBrowser")= "L"
  lcName    = "Rick Schummer's Font Changer"
  lcComment = "Developed by RAS for online forum discussion and example"

  IF TYPE("_oBrowser")= "O"
    * If Class Browser is running, use Addin() method
    _oBrowser.Addin(lcName, STRTRAN(SYS(16),".FXP",".PRG"), ;
                    "ACTIVATE", , , lcComment)
  ELSE
    * Use the low level access of the Browser registration table
    IF FILE(HOME() + "BROWSER.DBF")
      lnOldSelect = SELECT()

      USE (HOME() + "BROWSER") IN 0 AGAIN SHARED ALIAS curRASDateChanger
```

```
      SELECT curRASDateChanger
      LOCATE FOR Type = "ADDIN" AND Name = lcName

      IF EOF()
        APPEND BLANK
      ENDIF

      * Always replace with the latest information
      REPLACE Platform WITH "WINDOWS", ;
              Type     WITH "ADDIN", ;
              Id       WITH "METHOD", ;
              Name     WITH lcName, ;
              Method   WITH "ACTIVATE", ;
              Program  WITH LOWER( STRTRAN( SYS(16), ".FXP", ".PRG")), ;
              Comment  WITH lcComment
      USE

      SELECT (lnOldSelect)
    ELSE
      MESSAGEBOX("Could not find the table " + HOME() + "BROWSER.DBF" + ", ;
                 please make sure it exists.", ;
                 0 + 48, ;
                 _screen.Caption)
    ENDIF
  ENDIF

  RETURN
ELSE
  * Check to see if we really got called from the Class Browser
  IF !PEMSTATUS(toBrowser, "lFileMode", 5)
    RETURN .F.
  ENDIF

  * Now change the font
  toBrowser.SetFont("Tahoma", 8)
ENDIF

RETURN
```

The action side of the add-in just checks to see if the object passed to the procedure is indeed an instance of the Class Browser and calls the Class Browser *SetFont()* method with the font name and the font size as the two parameters.

Task List

The Task List tool is new in Visual FoxPro 7. It provides a mechanism to track a list of to-do items and tasks within the development environment. Initially one might think this is not such a big deal with all the nice task tracking capabilities of your favorite personal digital assistants (PDAs) or applications like Outlook. The importance of the Visual FoxPro Task List lies in the integration with the various editors in Visual FoxPro. You can add bookmarks in the editors. These bookmarks translate into shortcut tasks in the Task List. You can use the Task List to then open up the appropriate editor with the method or program available.

There are three types of tasks that are tracked by the Task List tool. Shortcuts are specific references to a line of code that you want to return to or want quick access to. User Defined Tasks are added through the Task List tool user interface. The “Other” tasks can only be added

to the list programmatically. They all track the same information in the task table (referenced with the new system variable called **_FoxTask**).

The Task List tool is really two separate items coupled together. The first is the Task List engine; the second is the user interface. Together they comprise the TASKLIST.APP, which is executed when you select the Task List option from the Tool menu.

The Fox Team at Microsoft has documented the Task List engine for Visual FoxPro developers to use. The entire programming interface can be found in the ARCHITECTURE.DOC file in the VFP\Tools\XSource\VFPSource\TaskList\Documentation folder. This folder and the rest of the source code for the Task List (and the other Visual FoxPro Xbase tools) is available if you unzip the XSOURCE.ZIP file installed in the Tools\XSource folder.

How do I add my own custom fields to the Task List? (*Example: MegaFoxTLExt.dbf/fpt*)

The Task List tool provides developers the capability to extend the data that is tracked for a task via custom fields. These fields are stored in a separate table with a one-to-one relationship to the main task list table. To access the field extension, right-click on the Task List and choose Options. There you can specify or create the "user-defined column table."

This custom field table must include a 10-character UniqueID field that is used to link the custom fields to the base fields. The UniqueID field is automatically included when you create a new table. All other fields are fair game. One recommendation we can make is to not name a custom field the same as one of the base fields (see **Table 3** for a complete list) to avoid confusion. Our testing revealed that all the Visual FoxPro data types are accessible from the Task List tool except for general, currency, and the two binary fields for general and memo. If you add one of the banished data types, the Task List engine will just ignore them.

***Table 3**. The base table contains the 11 fields for the developer to expose in the Task List user interface.*

Field name	Data type	Size
UNIQUEID	Character	10
TIMESTAMP	Numeric	10
FILENAME	Memo	4
CLASS	Memo	4
METHOD	Memo	4
LINE	Numeric	6
CONTENTS	Memo	4
TYPE	Character	1
DUEDATE	Date	8
PRIORITY	Numeric	1
STATUS	Numeric	1

The Task List Options dialog does not provide a mechanism to add additional fields to the current custom fields table. You can take control of this situation by using the trusty **MODIFY STRUCTURE** command and making the appropriate changes. The Task List tool will recognize the changes the next time it is started.

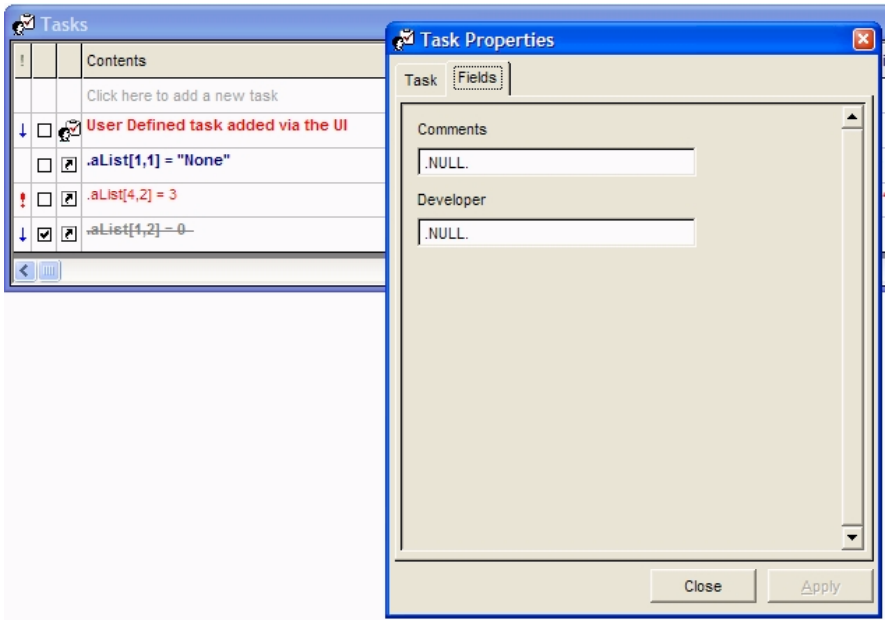


***Figure 19**. The Task Properties form is not very helpful in displaying the values stored in the custom properties.*

There are a number of extended fields we have added to our user-defined fields including developer (to differentiate team tasks), comments or instructions (to allow developers to type in comments about the task or instructions for other developers or a reminder for yourself), and project (see **Figure 19**).

The problem with these fields is that there is no hook in the Task List to provide default values, and since they are free tables, no mechanism to hook into the database engine. Later in this chapter we will discuss a solution to this issue.

How can I use my custom fields in the Visual FoxPro Task List?

Once you have established your own custom fields for the Task List, you need to add the columns to the user interface to be able to fill in the information, or open the Task dialog and check out the Fields page.

The Task List context menu has a Column Chooser option. Selecting this will bring up the Column Chooser dialog. The base fields and the user-defined fields are available to be added to the user interface. Only the fields not currently in the user interface are available. The position of the column will depend on the current active column. Select the column you want the new column to be added next to (to the right of the column selected).

Opening up the task (also started from the context menu) brings up the Task Properties, which has a second page that displays all the custom user-defined fields. We have found that this page is broken from a display perspective. All the fields default to the NULL value. Another thing is lacking is that all the fields are exposed in textboxes, even for memo fields. The page allows data entry and saves the changes, but the changes do not get displayed the

next time the page is displayed. If the column is in the Task List you can see the change made in the Task Properties dialog.

How can I add tasks programmatically to the Task List?

Earlier in this chapter we discussed how developers can extend the information tracked in the Task List and how there was not a built-in way to provide default values. One way to work around this limitation is to add records to the Task List programmatically. This can be accomplished using a reference to the Task List engine.

The Task List engine is referenced via the **_oTaskList** system variable. A task is a reference to an object created with a **SCATTER MEMVAR TO NAME**. To add a task you need to get a reference to a blank or empty task object. To get this reference you make a call to the Task List engine *GetTaskObject()* method. This will create the object reference with properties for each column (both base and custom) in the Task List. Each of the properties starts with an underscore. Make the assignments to these properties that you want as your default values. After all the properties are set you call the *AddTask()* method, passing the task object as the parameter.

```
IF VARTYPE(_oTaskList) = "O"
  * Grab a reference to it
  loTaskList = _oTaskList
ELSE
  IF NOT EMPTY(_tasklist)
    DO (_tasklist)
    loTasklist = _oTaskList
  ELSE
    ERROR "Task List application is not available in this session of VFP."
    RETURN .F.
  ENDIF
ENDIF

WITH loTasklist
   * Get an empty Task object
   loTask = .GetTaskObject()

   WITH loTask
      * User Defined Task
      * Assign the default values
      ._Type     = "U"
      ._Class    = SPACE(0)
      ._Method   = SPACE(0)
      ._Line     = 0
      ._FileName = SPACE(0)
      ._Contents = SPACE(0)
      ._DueDate  = DATE() + 1
   ENDWITH

   .AddTask(loTask)
ENDWITH

RETURN
```

The unique id assigned to the task is returned from the *AddTask()* method. You can use the unique id to update the record, position the record pointer in the FOXTASK.DBF, or locate records in the user-defined fields table.

One idea is to create a task to ship the test version to the quality assurance department each time a build is made. This could be accomplished by adding code to a projecthook *AfterBuild()* method. A similar concept is to add some tasks based on your project startup wizard.

How can I update tasks programmatically in the Task List?

The Task List tool provides two ways of natively updating tasks via the user interface. Similar to adding records, the Task List engine also provides a programmatic mechanism to updating records in the Task List.

The key to updating an existing record is getting a reference to the task. This is accomplished using the Task List engine *GetTask()* method. Passing the unique id for the task will return a task object if the task is found; otherwise, the *GetTask()* will return a logical false. Once you have a reference to the task you can change the "field" properties (underscore followed by column names) including both the base and user-defined columns.

```
LPARAMETER tcId

IF VARTYPE(_oTaskList) = "O"
  * Grab a reference to it
  loTaskList = _oTaskList
ELSE
  IF NOT EMPTY(_tasklist)
    DO (_tasklist)
    loTasklist = _oTaskList
  ELSE
    ERROR "Task List application is not available in this session of VFP."
    RETURN .F.
  ENDIF
ENDIF

* Use the parameter passed in to look up Task List record
loTaskData = loTaskList.GetTask(tcId)

IF VARTYPE(loTaskData) = "O"
   * Update the Due Date, Priority, and Status
   loTaskData._DueDate  = {02/20/2002}
   loTaskData._Priority = 2
   loTaskData._Status   = 1

   loTaskList.UpdateTask(loTaskData)
ELSE
   MESSAGEBOX("Task not found, could not be deleted.", ;
              4+48, ;
              "Update Task List Item")
ENDIF

RETURN
```

One idea to implement using this technique is to have code that updates the completed status, update a custom completed date column, and a custom completed developer column.

How can I delete tasks programmatically in the Task List?

The Task List Manager allows developers to delete tasks via the context sensitive menu. The Task List engine allows developers to delete tasks programmatically using the *RemoveTask()* method.

The steps to delete a task are to first get a reference to the Task List engine (`_oTaskList`). After you have a reference to the engine, you get a reference to the task you want to delete. It is not necessary to get a reference to the task, but it is good programming to check to see whether it exists before deleting it. This is accomplished by passing the unique identifier key to the Task List engine *GetTask()* method. If the task is found, the returned reference will be an object. If it is not found, *GetTask()* will return a false (`.F.`). The last step is to call the *RemoveTask()* method, passing the unique id for the task.

```
LPARAMETER tcId

IF VARTYPE(_oTaskList) = "O"
  * Grab a reference to it
  loTaskList = _oTaskList
ELSE
  IF NOT EMPTY(_tasklist)
    DO (_tasklist)
    loTasklist = _oTaskList
  ELSE
    ERROR "Task List application is not available in this session of VFP."
    RETURN .F.
  ENDIF
ENDIF

* Use the parameter passed in to look up Task List record
loTaskData = loTaskList.GetTask(tcId)

IF VARTYPE(loTaskData) = "O"
   lnResult   = MESSAGEBOX("Are you sure you want to delete the task?", ;
                           4+32, ;
                           "Delete Task List Item")

   IF lnResult = 6   && Yes
      loTaskList.RemoveTask(loTaskData._UniqueId)
   ENDIF
ELSE
   MESSAGEBOX("Task not found, could not be deleted.", ;
              4+48, ;
              "Delete Task List Item")
ENDIF

RETURN
```

If the unique id for the task cannot be found, the *RemoveTask()* method will just ignore the request. The deletion will be immediate. Tasks cannot be recalled through the user interface, but can be by recalling the record. There is no mechanism to pack the table, so the logically deleted records will remain in the FoxTask table until the developer takes it upon himself to `PACK` the file.

How can I fix a Task List when it seems to have lost its mind?

Occasionally the Task List will lose its mind. We have to remember that the Task List tool is a 1.0 release, and like most software with that version, there are quirks and bugs. Some of these bugs will disable the Task List tool from even starting and can easily make it unusable. We have a technique that restores some stability to a Task List that will not start, might not show all the tasks in the FOXTASK.DBF, or might just be crashing on "loTask variable not found" errors.

The tasks for the Task List are stored in a table called FOXTASK.DBF. The table used by the current Task List is stored in a VFP system variable called **_FoxTask**. You can open this table via a standard **USE** command and manipulate the records. We have found that adding and deleting records to this table without the Task List engine reference can be troublesome to the user interface. We have noticed that modifying the existing records for the due date and contents is not usually a problem, but have established some instability when making changes to the other base fields.

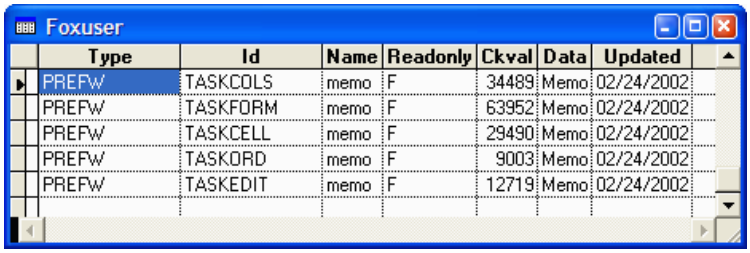
The configuration items that define some attributes of the Task List engine and the user interface are stored in the FOXUSER.DBF resource table. There are five records in the resource table (see **Figure 20**). Deleting the five records will force VFP to re-create the default settings and a new set of records in the resource table the next time the Task List is restarted. To delete the Task List records use the following code:

```
SET RESOURCE OFF
USE (SYS(2005)) EXCLUSIVE
SET FILTER TO "TASK" $ Id
BROWSE
```

Delete the five records and then:

```
PACK
USE
SET RESOURCE ON
```

Restart the Task List tool. You will need to reselect the User-Defined Column Table via the Options dialog to re-establish your extended columns and then reset the columns displayed via the Column Chooser dialog.



Type	Id	Name	Readonly	Ckval	Data	Updated
PREFW	TASKCOLS	memo	F	34489	Memo	02/24/2002
PREFW	TASKFORM	memo	F	63952	Memo	02/24/2002
PREFW	TASKCELL	memo	F	29490	Memo	02/24/2002
PREFW	TASKORD	memo	F	9003	Memo	02/24/2002
PREFW	TASKEDIT	memo	F	12719	Memo	02/24/2002

***Figure 20**. There are five records in the FoxUser.dbf that retain the settings and configuration of the Task List.*

What happens to the Task List tasks after I add an existing user-defined fields table?

One Task List feature available to the developer is being able to select an existing user-defined column table. There is one problem with this; all the tasks in the Task List disappear. They are not deleted from the FoxTask table, they just do not show up in the user interface. The problem is that the Task List establishes a one-to-one relationship between the base fields table (FoxTask) and your custom fields table. This relationship will likely not be correct with an existing table. The solution is to manually open both the FoxTask table and the user-defined table and add the needed records to the user-defined table.

Putting it all together with the G2 Task List Editor (*Example: TLEditor.scx/sct*)

Microsoft has put in place enough hooks to completely build our own Task List Editor. So we decided to put some of this newfound knowledge to work in a tool called the G2 Task List Editor and address some of the limitations of the Visual FoxPro Task List.

You can add (user-defined and other tasks), update, and delete tasks. We decided not to add support for shortcuts task types since it is much easier to add tasks from the various code editors. The data is "record buffered" since all the data is accessed via the task object data. It can be reverted until you move to another task. You can accomplish the same thing in the Visual FoxPro Task List if you edit the data in the Task Property dialog, but the "grid view" of the tasks has no way to revert any changes.

There are a couple of features that are included in the G2 version that are not included in the one that ships with Visual FoxPro (see **Figure 21**). The first issue addressed is that the native Task List does not display the existing information in the user-defined columns in the Task Properties. The example ships with _Developer and _Comments extended properties exposed. This version of the tool also provides a mechanism to have default values for the user-defined columns in the various add methods. The native tool does not have a mechanism to add "other" task types; this can be accomplished by clicking on the Add Other task button.

If you want to expose the custom Developer and Comment properties (extended fields exposed on the G2 Task List Editor), you need to add these user-defined columns to the Task List. If they are not in your list of custom fields, these properties are not exposed on the user interface. See the section "How do I add my own custom fields to the Task List?" in this chapter with steps to add your own custom fields. These fields are also available in the sample extended column table supplied in chapter downloads. See MegaFoxTLExt.dbf/fpt.

There is a known issue with the initial release. If you delete all the tasks, there are a number of problems with the G2 Task List Editor. It is causing the Content column of the grid to be truncated. Restarting the form will clear the problem. We are going to address this in a service pack. Check the Hentzenwerke Web site for updates.

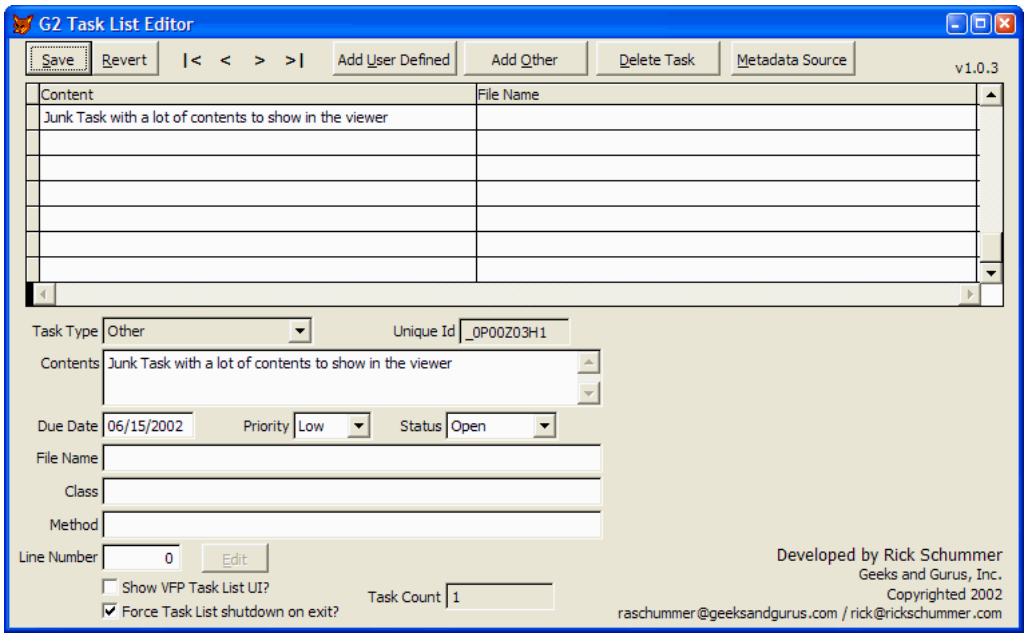


***Figure 21**. The G2 Task List Editor provides some functionality not available with the native Visual FoxPro Task List.*

Future enhancements to this tool include capabilities to pack the metadata tables, filtering specific task types, recalling deleted tasks, copying a list of tasks to the clipboard (so they can be copied to an e-mail), and generating some paper reports.

Object Browser

The Object Browser is new with Visual FoxPro 7. It exposes the public and protected interfaces of COM object libraries and ActiveX controls. Inside these libraries is a wealth of information concerning the properties, events, methods, constant values, and classes available for developers. This tool is very important to developers who write Automation code and need to understand the documented ways of using a particular Automation object.

How do I execute the Object Browser programmatically?

The Object Browser is available on the Tools menu. To start the Object Browser programmatically just execute the following code in the Command Window:

```
DO (_ObjectBrowser)
```

The system variable **_ObjectBrowser** contains the path and application name of the Object Browser. Like all of the Microsoft provided developer tools, the architecture is open and you can replace the entire tool if you see value in doing so.

How do I get rid of "cached" objects?

The Object Browser will cache the information displayed for an object to improve the overall performance of the Object Browser. It can take a significant amount of time to read all the properties, events, constants and so on when opening up a new object. For instance, it takes more than 50 seconds on our 450Mhz Pentium III to expand the Constants tab on the Excel object.

This information only changes when the object is upgraded by the manufacturer and reinstalled on the computer. To improve the performance of the second and subsequent times that the class is opened, the Object Browser stores the object information in a table (OBJECTBROWSER.DBF).

If you want to refresh the information and have the Object Browser read directly from the COM or ActiveX object, just press the refresh button on the Object Browser toolbar. The refresh button is the fourth from the left and looks like the Internet Explorer HTML page refresh button.

How do I determine the values of constants defined in a COM object?

One of the truly grueling tasks in developing Automation code is determining the constants used in the examples. These constants can be translated into **#DEFINE** code. Before we had the Visual FoxPro Object Browser we needed to trudge through Help files, hope examples documented the values, or use a tool like the Object Browser found in the VBA editors of Microsoft Office to find these values. This was a time-consuming process for sure. Tools like the West Wind GETCONSTANTS.EXE would read the type libraries and generate the **#DEFINE** code, which is easily compiled by Visual FoxPro.

The Object Browser can generate the **#DEFINE** code efficiently and is a real time-saver. To accomplish this, open up the COM or ActiveX component, and drill down the TreeView to expose the Constants node. Open up a program editor (program, or class method). Drag the Constant branch and drop it in the editor. Not only is the **#DEFINE** code typed in with the constant name and value, but the documentation for the constant is also included as a comment for the **#DEFINE** if the constant has a description. If you drag the constants branch you will get all the constants in the editor. You can also drag individual constants if you only need specific ones (see **Figure 22**).

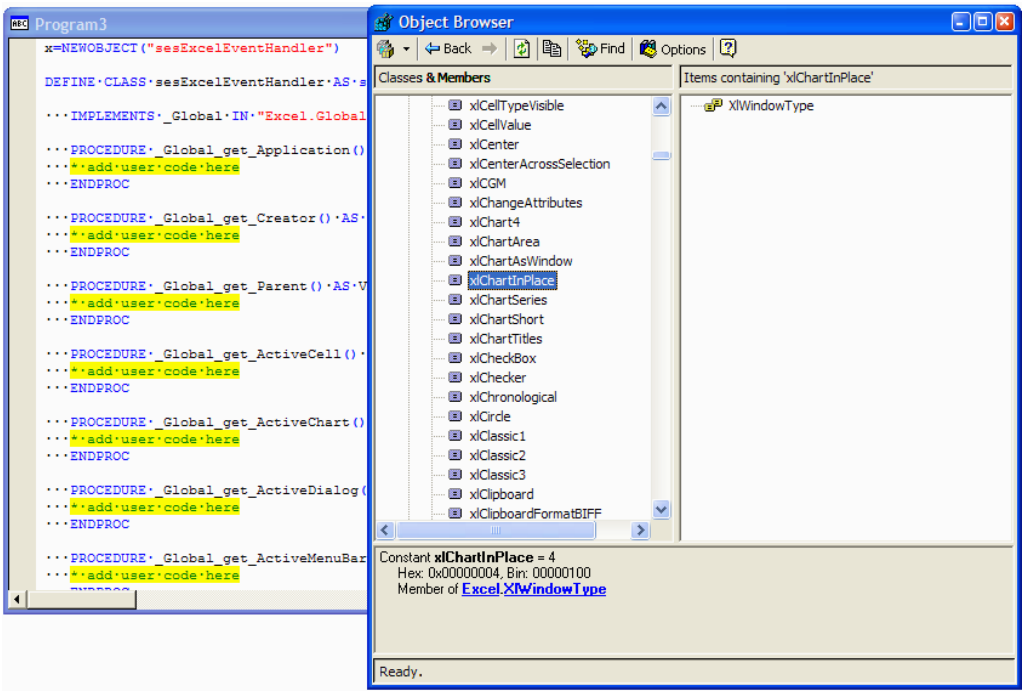


Figure 12.22 *The Visual FoxPro Object Browser quickly generates #DEFINES for constants and the template code for a class that implements event handling functionality.*

How can I use the Object Browser to create class templates to implement interfaces?

A very powerful new feature in Visual FoxPro is the capability to write code in our custom applications that responds to events in other applications. For instance, we can now write code to respond to a user closing a spreadsheet, or sending an e-mail in Outlook, or doing a mail merge in Word. This is done with the new **`IMPLEMENTS`** clause of **`DEFINE CLASS`** as well as the new **`EventHandler()`** function.

The Object Browser assists developers in writing tedious code in this respect. First, open up the COM or ActiveX control in the Object Browser. Then drill down through the TreeView and locate the Interfaces node. Open up a program editor (program, or class method). Drag the interface node and drop it in the editor. The class definition is written, including the **`IMPLEMENTS`** and template code for each of the methods that are exposed. All you have to do at this point is rename the class from MyClass to something more descriptive, and add code to the appropriate method.

How do I find out the name of the OCX file to ship with my deployment setup?

The new Object Browser helps Visual FoxPro developers with numerous features for ActiveX controls. One of the simpler, yet more helpful items is that it displays the actual file name for the OCX and other details about the control.

Open up the Object Browser and select an ActiveX control from the list. If you select the root node for the control, there are details about the OCX displayed in the bottom pane of the Object Browser. Information like the file name, the Help file, and the GUID is presented for the developer. This can come in handy when you need to find out what OCX file is to be included in a deployment package, and determine where the Help file is installed on the hard drive.

Project Manager

The Project Manager has been around for quite some time and is a user interface to the project files (PJX). It has changed over the years to increase developer productivity, and with Visual FoxPro 6 it added a COM interface to increase the ability to integrate with a new generation of developer tools. We dedicated an entire two chapters in KiloFox to the Project Manager tips. We still have one more up our sleeves for this chapter.

How can I automate the author settings in the Project Info dialog?

(*Example: PopulateProjectInfoDialog.prg*)

The Project Object introduced in Visual FoxPro 6 opened up new avenues for developers to manipulate the contents of the project files. Previous to this we needed to open up the PJX file and write code to make changes. There are a number of properties that can be set that appear on the Project Info dialog like Debug, Encrypted, HomeDir, and Icon. One glaring omission is the Author information.

This can be automated via a program that keyboards the information:

```
IF TYPE("_vfp.ActiveProject") = "O"
   IF WONTOP(_vfp.ActiveProject.Name)
      * Everything is ready, project is most active window
   ELSE
      * ACTIVATE WINDOW (JUSTFNAME(_vfp.ActiveProject.Name))
      ACTIVATE WINDOW ("Project Manager")
   ENDIF
ELSE
   RETURN
ENDIF

* Cannot automate the project menu since it is not a system menu
* The shortcut to the Project Info dialog is Ctrl+J, plus since
* the dialog opens on the page tab, you need to tab to the first
* textbox.
KEYBOARD '{CTRL + J}'
KEYBOARD '{TAB}'

* Author
KEYBOARD 'Richard A. Schummer'
KEYBOARD '{TAB}'
```

```
* Company
KEYBOARD 'Geeks and Gurus, Inc.'
KEYBOARD '{TAB}'

* Company
KEYBOARD '2921 East Jefferson Ave, Suite 300'
KEYBOARD '{TAB}'

* City
KEYBOARD 'Detroit'
KEYBOARD '{TAB}'

* State
KEYBOARD 'MI'
KEYBOARD '{TAB}'

* Country
KEYBOARD 'USA'
KEYBOARD '{TAB}'

* Postal Code
KEYBOARD '48207'
KEYBOARD '{TAB}'

RETURN
```

The code at the beginning of the program makes sure that there is a project open and brings it forward to be the active window. This is necessary because we want the Project menu activated so the shortcut key to the dialog is available. From that point on we are keyboarding the necessary keys to navigate from textbox to textbox and fill in the appropriate information.

Conclusion

One of the most powerful aspects of developing with Visual FoxPro is the ability to programmatically extend the interactive development environment and the various developer tools that ship with the base product. This chapter not only showed how you can hook into the base tools, but also demonstrated how you can write your own tools that extend the development environment.

Chapter 13
Working with Remote Data

It is an increasingly common requirement that Visual FoxPro applications should be able to access data that is not stored in local Visual FoxPro tables. Fortunately Visual FoxPro has always been a superb tool for accessing and manipulating remote data. In this chapter we will cover the issues associated with connecting to, and manipulating, remote databases with Visual FoxPro.

Running the examples

All of the examples in this chapter were written and tested using SQL Server 2000 (SP1). Unless otherwise stated, they all use a *System DSN* named "*SQLPubs*" that connects to a SQL Server installation on the local machine using the default name "*(Local)*". The System Administrator login *sa* is used, with a password of *sa* to log into the standard sample database *Pubs*. If you do not have these settings, and do not want, or cannot, change your configuration, you will need to modify the examples accordingly.

Connecting to remote data

One of the consequences of having to work with a remote database is that getting access to the data is no longer just of matter of opening a table or view with the `USE` command, or creating a cursor by running a local SQL statement. The first requirement is that we have to establish a connection to the data source, and there are two basic mechanisms available for doing this—ODBC and OLEDB.

Both are application programming interfaces (APIs) designed for accessing a wide range of data sources. ODBC is the older technology and is designed for accessing relational data (using SQL) in a multi-platform environment. The newer OLEDB is designed to provide access to all types of data in a Component Object Model (COM) environment. Thus OLEDB includes the capability to access relational data using SQL, but also defines interfaces for accessing other types of data.

Visual FoxPro provides us with a set of native functions that can be used with ODBC drivers directly to return a FoxPro cursor containing the results. However, it does not have a corresponding set of functions for working directly with OLEDB. Instead, we have to use ADO (ActiveX Data Objects) to return the results as an object.

The choice of which method you use will depend upon your specific circumstances and requirements. However, in the context of working with a remote relational database (which is our primary focus in this chapter), the benefits of being able to retrieve data in native cursor format generally outweigh the performance benefits that OLEDB offers, and so we will concentrate mainly on ODBC.

How do I connect to a database using ODBC? (*Example: ConODBC.scx*)

There are two types of connection that can be used: either a "connection string" or a "named connection," which uses a predefined Data Source Name (DSN).

Connection strings

A "string" connection passes all of the necessary information required to gain access to the database as an explicit character string. This has the benefit of not requiring anything to be set up on the end user's machine (or network), but requires that all of the information must be hard-coded (or at least stored in, and read from, a metadata table). This can be a problem if the connection parameters are likely to change, because either application code, or metadata, must also be changed, which usually requires developer input. Visual FoxPro's `SQLSTRINGCONNECT()` function is used to work with connection strings.

Using SQLStringConnect()

There are several ways of using this function to create a connection. The following example, illustrated in **Figure 1**, shows the minimum string required to connect to the standard SQL Server Pubs database:

```
lcString = "Driver=SQL Server;Server=(local);Database=pubs;UID=sa;PWD=sa"
lnConHandle = SQLSTRINGCONNECT( lcString )
```

`SQLSTRINGCONNECT()` can also be used with a named connection (DSN) that defines the Driver, Server, and Database information:

```
lcString = "DSN=SQLPubs;UID=sa;PWD=sa"
lnConHandle = SQLSTRINGCONNECT( lcString )
```

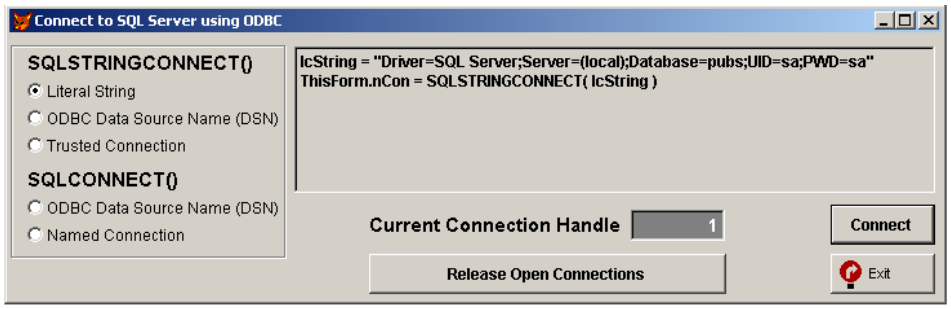


***Figure 1**. Connecting to remote data with ODBC (ConOdbc.scx).*

Notice that both of these mechanisms still require that the user ID and password be explicitly passed as part of the connection parameters. If your operating system supports it (and both Windows NT and Windows 2000 do), you can set up a "Trusted Connection" that uses the Windows login and password and so avoid the necessity of storing separate user ID and password information.

```
lcString = "DSN=SQLPubs;Trusted_Connection=Yes"
lnConHandle = SQLSTRINGCONNECT( lcString )
```

Named connections

The alternative to a connection string is to use a "DSN" (Data Source Name) to define the necessary parameters. DSNs are Windows entities that encapsulate the information required to create a connection and are stored in the system Registry. They can be maintained using the Windows ODBC Manager, and there are three types (see **Table 1**). Visual FoxPro's `SQLCONNECT()` function is used to work with DSNs.

***Table 1**. ODBC DSN types.*

DSN type	Description
USER	Local to the workstation and specific to the user who created it (that is, the DSN is related to the user ID).
SYSTEM	Local to the workstation, but may be accessed by any user who has access to the computer.
FILE	Can be stored anywhere and accessed by any user that has the relevant drivers installed.

Using SQLConnect()

This function is usually used with either a DSN, like this:

```
lnConHandle = SQLCONNECT( 'SQLPUBS', 'sa', 'sa' )
```

or with a Visual FoxPro connection object, as follows:

```
CREATE CONNECTION vfcon DATASOURCE 'SQLPUBS' ;
   USERID 'sa' PASSWORD 'sa'
lnConHandle = SQLCONNECT( 'vfcon' )
```

Note that whichever function you use to create the connection, a numeric "*handle*" will be returned. This value must be used to access the connection and so must be stored until the connection is released. Once you have a valid connection handle, you can use Visual FoxPro's built-in SQL Pass-Through (SPT) functions to work with the database.

To release an ODBC connection, simply call the `SQLDISCONNECT()` function, passing the handle for the connection that you want to close. (Note: Passing zero closes *all* open connections.)

How do I connect to a database using OLEDB? (*Example: ConOLEDB.scx*)

The answer to this one is, we are afraid, "It depends." The details of connecting using OLEDB depend upon the "provider" that you are using. While there are many providers available, including one for Visual FoxPro itself, they all fall into one of two categories: either product-specific or generic (ODBC). While the former may allow for extra settings or parameters, the generic provider is simply a wrapper around ODBC and so takes the same parameters as any other ODBC driver (see **Figure 2**).

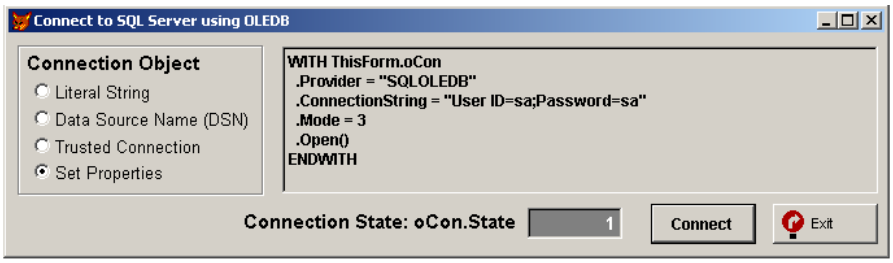


***Figure 2**. Connecting to remote data with OLEDB (ConOleDb.scx).*

The first requirement is to instantiate the connection object (this is done in the *Load()* method of the sample form):

```
oCon = CREATEOBJECT("adodb.Connection")
```

The connection object has an *Open()* method, which actually establishes the connection to the data source. The connection can be made using any of the methods described earlier for ODBC by passing the connection details as a parameter to the *Open()* method:

```
*** Define the connection string
lcString = "Driver=SQL Server;Server=(local);Database=pubs;UID=sa;PWD=sa"
oCon.Open( lcString )
oCon.Close()

*** Use a DSN
lcString = "DSN=SQLPubs;UID=sa;PWD=sa"
oCon.Open( lcString )
oCon.Close()

*** Use a trusted connection
lcString = "DSN=SQLPubs;Trusted_Connection=Yes"
oCon.Open( lcString )
oCon.Close()
```

or by setting the relevant properties on the connection object explicitly and then calling the *Open()* method without parameters:

```
oCon = CREATEOBJECT("adodb.Connection")
*** Set the connection properties explicitly
WITH oCon
  .Provider = "SQLOLEDB"
  .ConnectionString = "User ID=sa;Password=sa"
  .Mode = 3
  .Open()
  .DefaultDatabase = 'Pubs'
  .Close()
ENDWITH
```

A connection object has a *State* property that is set to 1 when connected, and 0 when not connected. To release an OLEDB connection, simply call the object's *Close()* method.

Connecting to a database that is not installed locally

There is little difference in the process of creating a connection when the database is not installed on your local machine. The only thing that really has to be done is to ensure that the connection string is constructed correctly so that the server can be located.

The Windows ODBC Manager automatically polls the network when you create a new connection and offers you a list of available servers as part of its user interface. Once you have created a DSN in this way, it is simple enough to get at the connection string using the methodology described in the section of this chapter on remote views (see the section "1. Configure the connection" later in this chapter).

If you have to connect using OLEDB, there is no standard equivalent to the ODBC Manager for creating the necessary connection strings. However, you can use Microsoft Data Link files to create or validate a connection string using an OLEDB Provider. The Knowledge Base article "Q195913: HOWTO: Generate ODBC & OLEDB Connection Strings with Data Links" gives full instructions. Note that if you are using Windows 2000 or later, you can download a utility that will create the necessary files. A link to that article ("Q244659: How to Create a Data Link File with Windows 2000") is included.

Basically, in order to connect a machine that does not have SQL Server installed locally to a database elsewhere on your network, two things are required. First, the data server has to be registered locally, and second, the code has to include the server name explicitly in the connection string. The Data Link Tool provides a user-friendly front end for creating the necessary entries in the Registry and generates a text file (with a .UDL extension) that contains the complete connection string. Once the server is registered, all that is needed is to amend the connection string to include the server name explicitly, like this:

```
oCon = CREATEOBJECT("adodb.Connection")
*** Set the connection properties explicitly
WITH oCon
  .Provider = "SQLOLEDB"
  .ConnectionString = "Data source=ACS-SERVER;User ID=sa;Password=sa"
  .Mode = 3
  .Open()
  .DefaultDatabase = 'Pubs'
  .Close()
ENDWITH
```

As noted in the Knowledge Base article, the same tool can be used to create ODBC connection strings. This is a great little utility, and we have to thank our Tech Editor, Steve Dingle, for bringing it to our attention.

Which is better, ODBC or OLEDB?

Unfortunately, there is no real answer to this question because the two technologies are not equivalent. It depends on the circumstances and your objectives. In the context of this chapter,

which is specifically concerned with accessing data in a remote SQL database, there is no real benefit in using OLEDB. We are using Visual FoxPro to access the data, so we have at our disposal a fast database engine with a fully functional SQL engine and powerful data manipulation language that is designed to work with cursors. ODBC retrieves data into Visual FoxPro cursors so we can work directly with them whereas, if we use OLEDB, we have to deal with ADO Recordsets. This is not difficult, but it does add a level of overhead that we don't incur if we use ODBC.

However, as soon as we have to deal with a non-SQL data source, the exact opposite is true. Even if we could connect to the data source using ODBC, we would not really be able to do much with it. It is in this scenario that OLEDB comes into its own. The remainder of this chapter is, therefore, based on the use of ODBC.

How can I be sure users have the correct settings? (*Example: DSNMgr.prg*)

This is a perennial problem when dealing with applications that require access to remote data, and the best advice we can offer is to ensure that your clients use a System DSN for your application (see Table 1 for the different types of DSN). This approach maintains a degree of security (since it requires that a user be logged in to their PC), but allows you to keep your code simple. You only need the DSN name, user ID, and password in your code; this information can easily be stored, and maintained, in a local Visual FoxPro table. There is no need to know about, or maintain, detailed information on server names, databases, drivers, and so on. Moreover, this approach should not mean any additional work for the client because, if a server configuration is ever changed, all DSNs that access that server have to be changed anyway.

The only problem with using a System DSN is that it does need to be set up on each user's local machine, but this can be checked programmatically, and you can even create a DSN programmatically if you have the necessary details available. All that is required is to create a few entries in the system Registry.

Registry structure for System DSNs

Information about System DSNs is stored in two sub-keys of the HKEY_LOCAL_MACHINE "root key. The first, software\odbc\odbcinst.ini, has one sub-key for each driver that lists the settings for that driver. There is also a single sub-key, named ODBC Drivers, which lists all installed drivers and could be used to generate a pick-list of available drivers (see **Figure 3**).

The second key, named software\odbc\odbc.ini, has one sub-key for each DSN that has been defined. Each sub-key has entries for the DSN parameters and their values. To create a new DSN, all that is needed is to create and populate the appropriate sub-key (see **Figure 4**). An additional sub-key, named ODBC Data Sources, contains the list of defined DSNs, and this is used to populate the native ODBC Manager dialog.

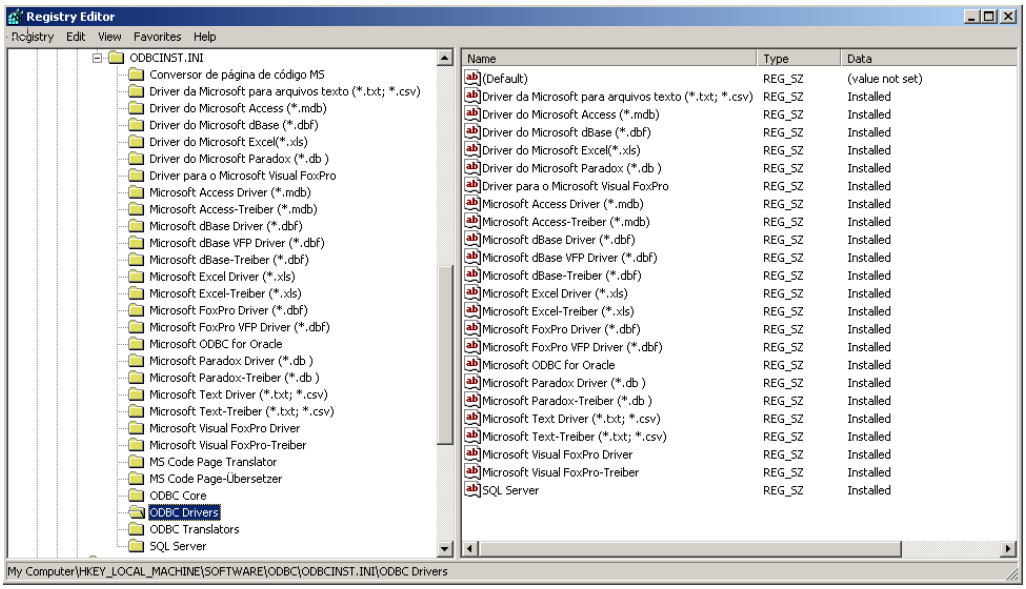


***Figure 3**. Installed driver list in the Registry.*

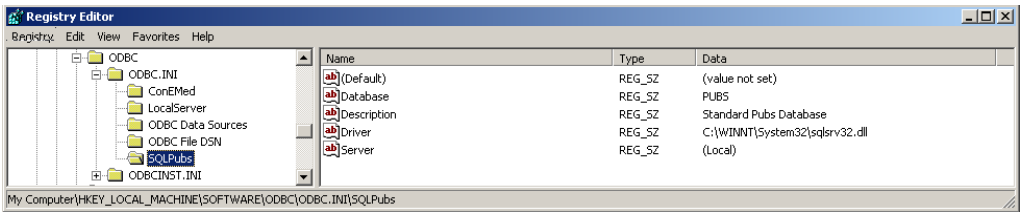


***Figure 4**. DSN definitions in the Registry.*

If you want the DSN that you define to appear in the ODBC Manager, an entry must be made under the ODBC Data Sources *sub-key. However, this is not actually required because the DSN will still work without one; it just does not appear in the Windows dialog. The default behavior of the DSNMgr class is to create this entry so that all DSNs will appear in the ODBC Manager.*

Creating DSNs programmatically with the DSNMgr class

This class is designed to handle the management of DSNs on user machines and uses two tables to store the metadata needed for validating and creating a System DSN at run time (see **Table 2**).

***Table 2**. Structure for DSNMgr metadata tables.*

DSNSetup.dbf			
iDSNPK	I	4,0	Primary Key
cDSNName	C	20,0	DSN Name
cDSNDesc	C	60,0	Description for this DSN
cDSNDriver	C	60,0	Name of the ODBC Driver to use (as stored in the Registry)
cDSNServer	C	40,0	Name of the Server to which the DSN connects
cDSNDBase	C	40,0	Name of the Database to use as default on the specified server
DSNConn.dbf			
iConPK	I	4,0	Primary Key
cConName	C	20,0	Connection Name
cConDSN	C	60,0	Name of the DSN used by this connection
cConUID	C	60,0	User ID for connection log-in
cConPwd	C	40,0	Password for connection log-in

Metadata

The first table, DSNSETUP.DBF, is used to define the minimum requirements needed to set up a DSN (that is, Name, ODBC Driver, and Server Name, and also allows for a default database and a description, which will be visible in the Windows ODBC Administrator). The second table, DSNCONN.DBF, is used to define "user connections." This reason for this structure is that it allows us to use the same DSN with different combinations of user ID and password, so that different users can be assigned to specific groups, or roles, on the server. An additional benefit is that this table can be used to manage the actual log-in process because it maps the user ID and password to a DSN name. All that we need to know in our code is the name of the "user connection" to use.

Methodology

The *DSNMgr* class is based on the *Session* base class so that it can have a private datasession where the metadata tables get opened. The *Init()* method handles opening the tables and also creates, and assigns to a property, an instance of our *Registry Manager* class that encapsulates the Windows API calls used to read from, and write to, the Registry.

The work is done by calling the custom *IsConnValid()* method with the name of a connection. This connection name is used to find the DSN in DSNCONN.DBF. Next, the Registry's "ODBC Data Sources" list is checked for the specified DSN. If found, *IsConnValid()* returns True. If the DSN is not found, the default behavior is to try and create it using the information from DSNSETUP.DBF. This behavior can be suppressed, if necessary, by passing any value that Visual FoxPro does not evaluate as empty to the *IsConnValid()* method as a second parameter.

Example

The download code for this chapter includes a database named DsnData that contains the metadata tables used by the DSN Manager. These tables contain the data for creating the standard SQLPubs connection that we have used throughout, as shown in **Table 3**.

***Table 3**. Data for DSNMgr example.*

DSNSetup.dbf		DSNConn.dbf	
Field	**Value**	**Field**	**Value**
iDSNPK	1	iConPK	1
cDSNName	SQLPubs	cConName	CH13Demo
cDSNDesc	System DSN for SQL Pubs Database	cConDSN	SQLPubs
cDSNDriver	SQL Server	cConUID	sa
cDSNServer	(Local)	cConPwd	sa
cDSNDBase	PUBS		

To run the example, instantiate the *DSNMgr* class and call the *IsConnValid()* method with the name of the connection, as follows:

```
loDM = NEWOBJECT( "DSNMgr", "DSNMgr.prg" )

*** To ensure that the DSN is created if not already present
*** Use this form:
llOK = loDM.IsConnValid( 'CH13Demo' )

*** To prevent the creation of the DSN if it does not exist
*** Use this form:
llOK = loDM.IsConnValid( 'CH13Demo', .T. )
```

How do I use remote views in Visual FoxPro?

Visual FoxPro's remote views provide a quick and easy way to access remote data without the necessity of worrying about the intricacies of using SQL Pass-Through. However, there is nothing "magic" about remote views; they are simply wrappers around SQL Pass-Through and use ODBC to connect to the server. There are, therefore, some limitations in the available functionality and a performance overhead associated with using them. For these reasons, our personal preference is to avoid using them in production code and to use cursors instead, even though that means that we have to do much more of the detailed management work ourselves.

Having said that, we must also point out that there are many examples of applications developed using remote views that work perfectly well. As always in Visual FoxPro, you must pick the approach that best suits your needs.

The following sections describe the four steps needed to configure your system and create a simple form using a remote view.

1. Configure the connection

The first requirement for using remote views in Visual FoxPro is to define a "connection." In this context a connection is a database object that manages and controls access to the ODBC connection (whether it is defined as a DSN or a connection string), thereby providing a simple interface to the back-end server. However, since they are database objects, a Visual FoxPro database container is needed to create one and to gain access to the visual designer that provides the basic setup for a connection (see **Figure 5**). Alternatively, the `CREATE CONNECTION` command and `DBSETPROP()` function can be used to create and fine-tune connection objects.

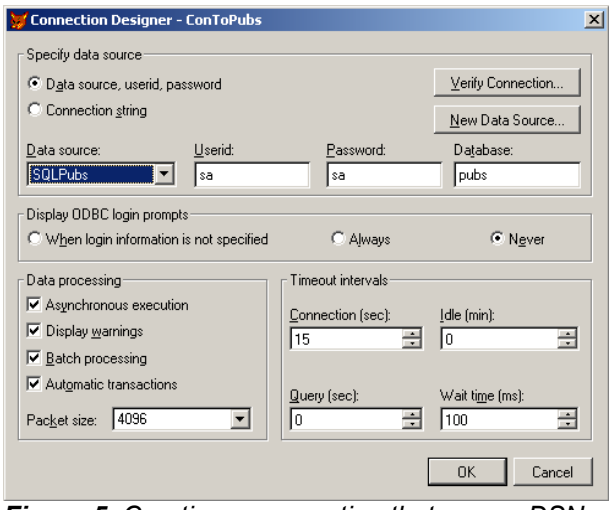


***Figure 5**. Creating a connection that uses a DSN.*

The only difficult part of connection design is that there are so many properties to set, all of which impact performance. The benefit of the Connection Designer is that it presents the important properties so that nothing gets forgotten. Using the designer we can create an ODBC connection that uses either a DSN or a connection string.

In fact, you can actually use the Connection Designer to create a connection string for you. When the option "connection string" is chosen, the combo and text boxes in the "Specify data source" section are replaced by a single textbox and button (see **Figure 6**).

That button displays a data source selection form that lists all the currently available data sources. When a source is chosen, Visual FoxPro will try to connect to it (you will be prompted for the user ID and password) and, if the connection succeeds, it will return the complete connection string for you.

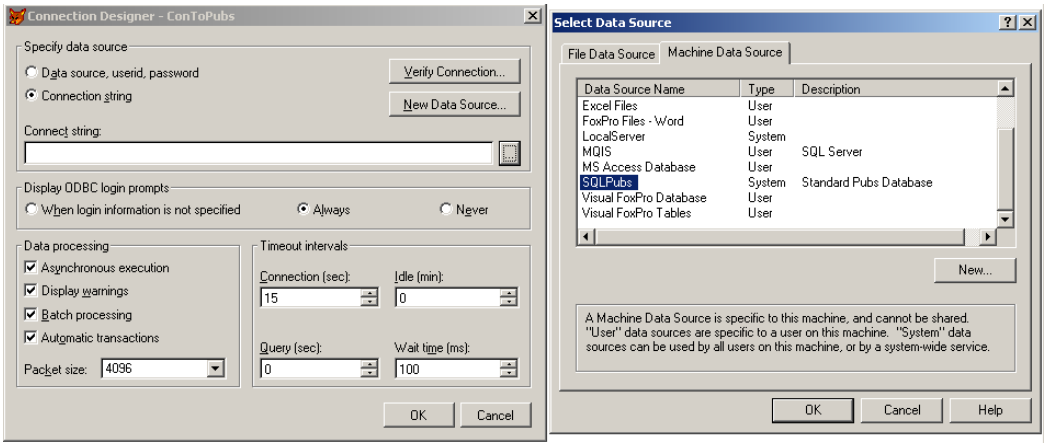


***Figure 6**. Visual FoxPro Connection Designer.*

Using this dialog, and choosing SQLPubs as the DSN, we get the following connection string generated:

```
DSN=SQLPubs;Description=Standard Pubs Database;UID=sa;PWD=sa;
    APP=Microsoft (R) Visual FoxPro;WSID=ACS-SERVER;DATABASE=pubs
```

Notice that in addition to the minimum necessary information, Visual FoxPro has included both the source application name and the workstation ID.

Irrespective of how they are created, ODBC connections are configurable, and **Table 4** describes the options available. These items can be accessed, and most can be set individually, using the native Visual FoxPro functions `SQLGETPROP()` and `SQLSETPROP()`.

***Table 4**. ODBC connection properties.*

Property	Description
Data Source UserID Password	The Data Source is the DSN that was created in the ODBC administrator. You can also use a connection string instead of a DSN. UserID and Password must map to a pre-defined server login.
Connection String	A connection string (sometimes known as a DSN-less connection) specifies the exact parameters needed to connect to a data server. For SQL Server, a typical connection string appears like the following: "Driver = SQL Server; Server = mySQLServer; UID = myUserName; Pwd = mysecret; APP = appname" The advantage to a connection string is that you do not need a pre-defined DSN. This does have the benefit that you need not worry about ODBC data source definitions. The drawback, of course, is that the connection string is hard-coded.
Display ODBC Login Prompts	This option can be used to prompt the user for their user name and password when a server connection is attempted. However, this is generally not a good idea, since a typical client/server application acquires server connections more than once during the life of the application. This setting should always be set to “Never” before deploying.
Asynchronous Connection	This option does not affect how remote views operate, since they always work asynchronously. However, it does impact the operation of SPT statements—which will not wait for the back end to complete when this is set to True (the generally preferred option).
Display Warnings	A warning is a message that generally makes sense only to a developer at design time. This option should normally be set ON during design, but turned OFF before deploying.
Batch Processing	This option does not affect remote view operation, since views can issue only one SQL statement at a time. However, SPT statements can be run in batches, so this setting can greatly affect the operation of SPT.
Automatic Transactions	A transaction is a single unit of work on a database. Therefore, if your view or SPT statement updates several records, this ensures that they are all committed successfully. Any failure on update will roll back all records. By turning this off, you must manually complete transactions. This has nothing to do with Visual FoxPro’s native transaction processing, but has everything to do with transactions on the data server.

***Table 4**. Continued.*

Property	Description
Packet Size	This allows you to specify the size of the data packets sent between the client and data server. However, it is highly likely that your network protocols will essentially ignore this value. It would be wise to leave this setting alone unless you can talk to your system administrator and verify that changing packet size would have any effect at all.
Connection timeout	This answers the question "How long will Visual FoxPro attempt to acquire a server connection before giving up?" Therefore, if the server is off-line or otherwise unavailable, Visual FoxPro will return an error only after this time has expired. The default is to wait 15 seconds.
Idle timeout	This setting allows idle connections to be automatically dropped by the server after the specified amount of time. It sounds like a good idea to set this to something greater than zero. However, there are many situations that require a static server connection (any resources are dropped when the connection drops), and an idle timeout could cause trouble in these situations.
Query timeout	This answers the question "How long will Visual FoxPro wait for a query to start returning data?" If the server is too busy processing other tasks, or the submitted statement produces a "run away" query, this timeout value will come into effect. Leaving it at the default of zero is not recommended since the aforementioned scenarios will hang the client, forcing abnormal shutdowns of Visual FoxPro.
Wait time	This property determines how often Visual FoxPro checks to see whether the server is still working or not. This should not be set too low because of the adverse effect on client performance.

2. Configure the remote data handling

Before using remote views you should also review Visual FoxPro's remote data handling, which has its own page in the Options dialog (under the Tools pad on the main system menu). Many of these settings can be altered for a specific view at run time, by using the `DBSETPROP()` function, but others can only be set through this dialog (see **Figure 7**).

Notice that although this dialog is for setting up the "Remote Data" options (see **Table 5**), the bottom half is actually concerned with the connection properties rather than data. It allows you to set your own default values for all of the properties that we covered in the preceding section.

In addition to the properties extracted into the Options dialog, there are additional items that can be set for a remote view by using either the `CURSORSETPROP()` function directly or through the Advanced Options dialog (which you can find under the Query menu in the View Designer). The Help file does provide a good account of these properties, but the ones detailed in **Table 6** are worthy of specific mention.

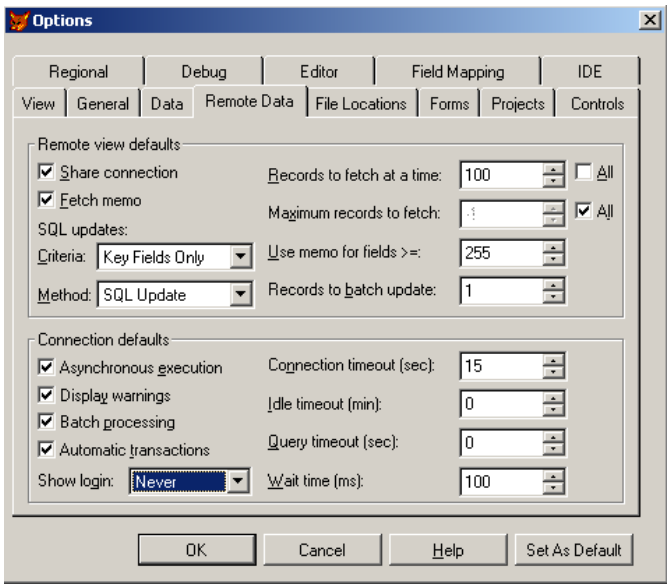


***Figure 7**. Visual FoxPro remote data configuration.*

***Table 5**. Visual FoxPro primary remote data handling settings.*

Property	Description
Share Connection	Unless you have a specific reason to do otherwise, set this to “Share”. This will prevent each view establishing its own connection to the server, which, if you are licensed by connection, can rapidly eat up your license as well as being heavy on network resources.
Fetch Memo	This property is ON by default, meaning that the contents of the Memo and General fields are downloaded with the records. By turning this property off, you will only download these fields when they are requested. However, this will keep your connection busy and unable to be shared until all the data has been downloaded to the client. It is generally better left on.
Records to Fetch at a time	Implements “progressive fetching.” This determines the number of records that Visual FoxPro will wait for before returning control back to your code. The default is 100, which means that all records after the first 100 are fetched by Visual FoxPro “in the background” (unless the “Fetch remote data as needed” property is set to true). If you always want the entire result set before your code continues, set this property to -1.
Maximum Records to Fetch	This is simply a limit on the number of records returned by the view. If you are dealing with very large record sets (many millions of records), it would make sense to set this property to avoid ever trying to retrieve them all. Otherwise, it is pretty safe to leave it at -1, which means all rows may be fetched. Note that the entire result set is still generated on the server; this simply limits how many records can be downloaded from the server into a single result set.
Use Memo for fields >=	If a field exceeds the number of specified characters, Visual FoxPro will create a memo field to hold the data. This can be used to increase the response time of a view, but only when used in conjunction with the “Fetch Memo” and “Fetch remote data as needed” properties. Be aware that this can tie up connections for long periods of time.
Records to Batch Update	When your updates are sent to the back end, they are sent individually, although you may have several records’ worth of updates. This property allows you to send more records with an update, but this property isn't supported by some servers; you should test this property against each remote server before deploying it in your application.

***Table 6**. Additional remote data settings.*

Property	Description
FetchAsNeeded Only fetches additional data when the client requests it	By default, this property is off, which means that Visual FoxPro will automatically download all data that the server provides in response to a query. When set, several of the aforementioned properties change their behavior because data will only be downloaded when it is specifically requested. For example, the number of records to fetch will no longer fetch records in the background; instead, it will download the first hundred and wait until a part of the next hundred is accessed before downloading the next hundred. However, when ON, this will tie up the connection and, unless you have a specific reason for doing so, should not normally be used.
CompareMemo Include memo fields in WHERE clause	Some data servers do not support the comparison of memo fields with their server-side counterparts. The result is a view that reports pending changes when there are none. Toggling this property can solve this problem.
Prepared Precompile SQL on back-end server	Precompiled statements tend to work more efficiently since the server will have preprocessed the SQL statement ahead of time. This allows the server to design a plan for executing the statement. However, certain back ends cannot take advantage of this feature, and setting it to True in those cases will actually slow performance.
SendUpdates Determines whether a remote view updates source data	By default this is set to False, but the setting of this property is crucial when working with remote views. Unless set to True, updates will not be sent to the back end. In the View Designer it is controlled by a checkbox on the Update Criteria tab, but it can also be set explicitly in code.

3. Define a remote view

To create a new remote view, simply open a database and choose "New Remote View" from either the pop-up menu or the Database pad of the system menu. You will notice that the first thing that happens is that the Select Connection dialog is displayed (see **Figure 8**). When creating a remote view, you must always specify which connection that view is going to use. This connection name is stored with the view as a read-only property and is available through `CURSORGETPROP()`.

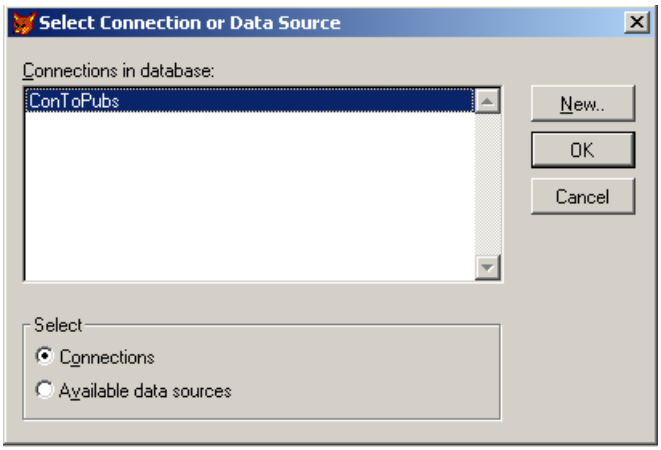


***Figure 8**. Connection selection dialog.*

Before Version 7.0, a remote view could only ever use the connection with which it was created. This was a *major* limitation on views, and one of the most welcome enhancements in Visual FoxPro 7.0 was, therefore, the extension of the **USE** command to support an additional **CONNSTRING** clause. This allows a view to be opened using a specific connection string instead of its pre-defined connection object. Note that although this only allows for a connection string, we can, as we have seen already, still use a DSN name as part of that connection string.

All of the following formats will work for opening a remote view named "rv_test":

```
*** Using the predefined connection object
USE rv_test

*** Using a connection string with a DSN
USE remviews!rv_demo1 CONNSTRING "DSN=SQLPubs;UID=sa;PWD=sa"

*** Using a fully qualified connection string
lcString = "Driver=SQL Server;Setup=C:\WINNT\System32\SQLSRV32.dll;
           Server=(local);Database=pubs;UID=sa;PWD=sa"
USE remviews!rv_demo1 connstring (lcString)
```

Once a connection has been selected, the standard Visual FoxPro View Designer is displayed (see **Figure 9**); the only difference is that the list of available tables is now coming from your back-end server via the ODBC link that is defined by the connection that you just specified. The process of defining the remote view is, from this point on, identical to that for defining a local view. Just as with a local view, you must not forget to check the Send SQL Updates box on the Update Criteria tab if you want the view to update its source tables.

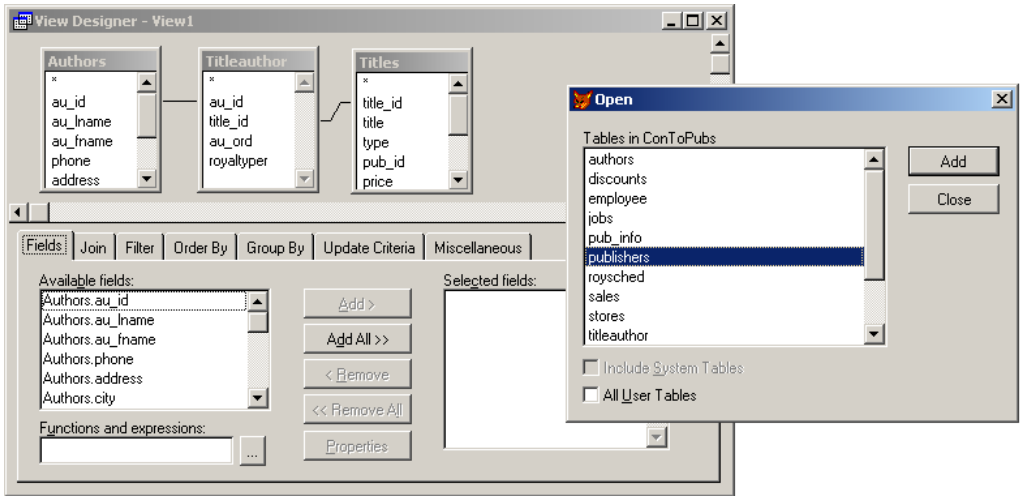


***Figure 9**. Creating a remote view.*

Remote views can also be created in code using the **CREATE SQL VIEW** command. The problem with doing so is that, as with local views, you may need to set a lot of properties directly. The sample database REMVIEWS.DBC includes a connection named ConToPubs and a simple view named rv_authors that is used in the sample form that follows. The code required

to create this view, generated using GENDBC.PRG but with repetition of default settings removed, is:

```
CREATE SQL VIEW "RV_AUTHORS" REMOTE CONNECT "ConToPubs" AS ;
      SELECT Authors.au_fname, Authors.au_lname, Authors.address,;
             Authors.city,  Authors.phone, Authors.state, ;
             Authors.zip, Authors.contract, Authors.au_id ;
        FROM dbo.authors Authors ;
    ORDER BY Authors.au_lname, Authors.au_fname

DBSetProp('RV_AUTHORS', 'View', 'UpdateType', 1)
DBSetProp('RV_AUTHORS', 'View', 'WhereType', 1)
DBSetProp('RV_AUTHORS', 'View', 'FetchMemo', .T.)
DBSetProp('RV_AUTHORS', 'View', 'SendUpdates', .T.)
DBSetProp('RV_AUTHORS', 'View', 'Tables', 'dbo.authors')
DBSetProp('RV_AUTHORS', 'View', 'Comment', "")

* Props for the RV_AUTHORS.au_fname field.
DBSetProp('RV_AUTHORS.au_fname', 'Field', 'KeyField', .F.)
DBSetProp('RV_AUTHORS.au_fname', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.au_fname', 'Field', 'UpdateName', 'dbo.authors.au_fname')
DBSetProp('RV_AUTHORS.au_fname', 'Field', 'DataType', "C(20)")
* Props for the RV_AUTHORS.au_lname field.
DBSetProp('RV_AUTHORS.au_lname', 'Field', 'KeyField', .F.)
DBSetProp('RV_AUTHORS.au_lname', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.au_lname', 'Field', 'UpdateName', 'dbo.authors.au_lname')
DBSetProp('RV_AUTHORS.au_lname', 'Field', 'DataType', "C(40)")
* Props for the RV_AUTHORS.address field.
DBSetProp('RV_AUTHORS.address', 'Field', 'KeyField', .F.)
DBSetProp('RV_AUTHORS.address', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.address', 'Field', 'UpdateName', 'dbo.authors.address')
DBSetProp('RV_AUTHORS.address', 'Field', 'DataType', "C(40)")
* Props for the RV_AUTHORS.city field.
DBSetProp('RV_AUTHORS.city', 'Field', 'KeyField', .F.)
DBSetProp('RV_AUTHORS.city', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.city', 'Field', 'UpdateName', 'dbo.authors.city')
DBSetProp('RV_AUTHORS.city', 'Field', 'DataType', "C(20)")
* Props for the RV_AUTHORS.phone field.
DBSetProp('RV_AUTHORS.phone', 'Field', 'KeyField', .F.)
DBSetProp('RV_AUTHORS.phone', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.phone', 'Field', 'UpdateName', 'dbo.authors.phone')
DBSetProp('RV_AUTHORS.phone', 'Field', 'DataType', "C(12)")
* Props for the RV_AUTHORS.state field.
DBSetProp('RV_AUTHORS.state', 'Field', 'KeyField', .F.)
DBSetProp('RV_AUTHORS.state', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.state', 'Field', 'UpdateName', 'dbo.authors.state')
DBSetProp('RV_AUTHORS.state', 'Field', 'DataType', "C(2)")
* Props for the RV_AUTHORS.zip field.
DBSetProp('RV_AUTHORS.zip', 'Field', 'KeyField', .F.)
DBSetProp('RV_AUTHORS.zip', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.zip', 'Field', 'UpdateName', 'dbo.authors.zip')
DBSetProp('RV_AUTHORS.zip', 'Field', 'DataType', "C(5)")
* Props for the RV_AUTHORS.contract field.
DBSetProp('RV_AUTHORS.contract', 'Field', 'KeyField', .F.)
DBSetProp('RV_AUTHORS.contract', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.contract', 'Field', 'UpdateName', 'dbo.authors.contract')
DBSetProp('RV_AUTHORS.contract', 'Field', 'DataType', "L")
```

```
* Props for the RV_AUTHORS.au_id field.
DBSetProp('RV_AUTHORS.au_id', 'Field', 'KeyField', .T.)
DBSetProp('RV_AUTHORS.au_id', 'Field', 'Updatable', .T.)
DBSetProp('RV_AUTHORS.au_id', 'Field', 'UpdateName', 'dbo.authors.au_id')
DBSetProp('RV_AUTHORS.au_id', 'Field', 'DataType', "C(11)")
```

As you can see, just as for local views, it is easier to use the Designer whenever you can!

4. Create the form (*Example: RVForm.scx*)

In practice there is nothing different about a form that uses a remote view rather than a local table or view except that, because remote views *must* use optimistic locking, the form has to be set up accordingly. The example is a perfectly normal Visual FoxPro form in all other respects. It was set up to use a private datasession and optimistic table buffering. The view rv_authors is opened automatically when the form is instantiated because it was added to the dataenvironment, just like a local view.

Note that we have opted to use table buffering in this example. Unless you have a very special reason for not doing so, this should be the normal state for any view that updates the back end directly. It is imperative that we keep the number of round trips between client and server to a minimum (if only for performance reasons). By using table buffering with the view, we require only one trip to the server to get the data, and updates are sent only when the user explicitly initiates the Save.

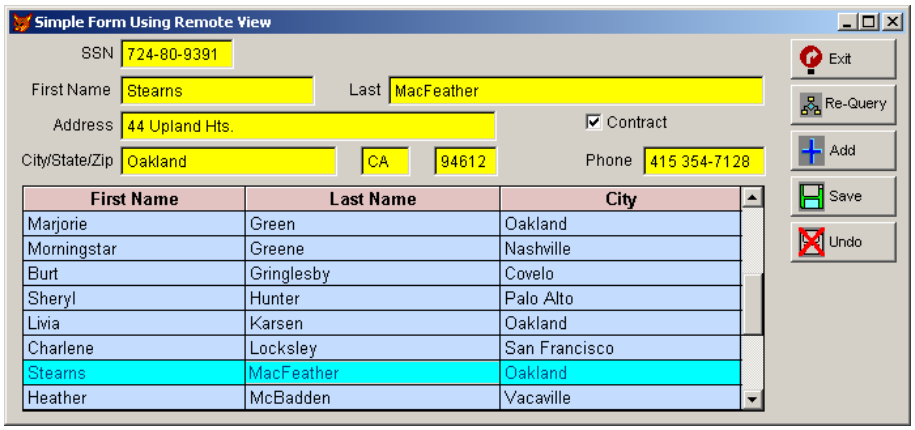


***Figure 10**. Simple data entry form using a remote view (RvForm.scx).*

For example, in the form shown in **Figure 10**, the updateable view displayed in the read-only grid is also used to populate the updateable entry fields. If we had used row buffering, then every time the user navigated in the grid, any changes made would be immediately sent to the back-end server. This would not be good design! The remote view has been added to the dataenvironment of the form and is populated with all data from the server automatically when the form loads. While this is very simple, it is not necessarily the best way to use a remote view—particularly if the data volume is large. One option is to use the *NoDataOnLoad* property to ensure that the view is opened (so that controls may bind to it) but no data is brought down until the view is explicitly queried.

The functionality in the form is simple enough and, as we noted earlier, is really no different from that required when using a local view. Note that because we are using table buffering, merely appending records to the remote view (which is all that the code in the form's *Add()* method does) has no impact for the data on the server until we issue an explicit **TABLEUPDATE()**. This is done with the following code in the form's *Save()* method:

```
IF ! TableUpdate(2, .F., 'rv_authors' )
    MESSAGEBOX('Unable to Save Changes, Retry or Cancel', 16, 'Major Waaah!')
ELSE
    MESSAGEBOX('Changes saved successfully to SQL Server', 64, 'Hooraaayyy!')
ENDIF
ThisForm.RefreshForm()
```

Of course, in a real application we would need to be a little more sophisticated in our error handling, but the principles do not change.

The Undo button allows us to revert changes in the view, but again, this will not affect the back-end server in any way. The server is only affected by **TABLEUPDATE()** commands. Here is the code from the *Undo()* method of the form:

```
*** Not much can go wrong with a TableRevert!
TableRevert(.T., 'rv_authors')
IF EOF( 'rv_authors' )
    GO BOTTOM IN rv_authors
    ThisForm.grdAuList.SetFocus()
ENDIF
ThisForm.RefreshForm()
```

Finally, we have a Re-Query option. This forces a query to be sent to the server in order to re-build the view from whatever data the server currently has in its tables. The code is simple enough:

```
*** Get rid of any uncommitted changes
TABLEREVERT(.T.)
*** Now Re-Query the view
REQUERY( 'rv_authors' )
ThisForm.grdAuList.SetFocus()
ThisForm.Refresh()
```

> ! *Note that before we can do the Re-Query we must ensure that we have no uncommitted changes in the view. If we were to have changes, Visual FoxPro would generate an error. It is to avoid this problem that remote views are row-buffered by default, so, whenever you change to table buffering, it is important to ensure that you do not issue a Re-Query while the view's buffer contains pending changes.*

Summary

Remote views provide a quick and convenient way to manage data on a remote server. More importantly, Visual FoxPro has good visual tools that we can use to configure the settings for

connections and for basic access to remote data. However, views have many limitations and, in our opinion, are not the best vehicle for developing major client/server applications.

What's wrong with remote views?

The short answer to that question is that nothing is "wrong" with remote views. The problem is that, as demonstrated in the next section ("What should I use instead of remote views then?"), remote views have some serious limitations when compared to using SQL Pass-Through (SPT) directly. In fact, even if you can use remote views initially, you will almost certainly end up having to use SPT sooner or later, so that you can access stored procedures or control transactions on the server.

Notwithstanding the preceding comments, remote views do suffer from one serious drawback that makes building flexible client/server applications with them much more difficult. They have to be pre-defined, which means that there is no simple way of using them to deal with "ad-hoc queries." Consider the following scenario.

You need to retrieve, from your data server, the names, addresses, and contact details of all customers who live in the state of Ohio. This is a simple enough query, and we could easily create the necessary remote view to get this data. In fact, we would probably construct it as a parameterized view, so that we could pass the state code in at run time and retrieve the data for any state, not just Ohio. This is exactly the sort of thing that remote views are good at.

However, if we now want the names, addresses, and contact details of all customers who are between the ages of 35 and 50 and who have not bought anything from us in the past six months, irrespective of where they live, we have a problem. The output fields, source tables, and joins are identical to the preceding query, but there is no way that we can reuse the parameterized view we just created to get this data. We must either create an entirely new view, re-define the existing view, or create a very complex parameterized view to begin with, and manage it in our code.

This is what we mean by an ad-hoc query; essentially it is a query where the filter condition may vary at run time depending on user actions. It is this sort of query that remote views are *not* good at.

Why does it matter? The form illustrated in **Figure 11** is an example of the type of search form that we regularly see in client/server applications. The user can specify no criteria at all (in which case all data is returned), or any combination of criteria that they wish to specify. This type of search functionality is almost always required in an application, and in client/server applications, where the volumes of data are normally large, it is essential to be able to allow the user to specify a subset of data for review. After all, we really do not want to bring down many thousands of records into a grid every time that the user needs to choose the particular one they want! In this sort of form, the filter clause for the query is built dynamically and then executed to populate a standard cursor to which the result grid is bound. Typically this cursor will include all necessary primary keys for dependent tables. When the user chooses a record, the necessary sub-queries can be formulated using the correct keys to retrieve only the specific data that is needed at that time. The example form FRMACCTS.SCX, included in the download code, that is used to illustrate the data classes later in this chapter implements precisely this methodology.

Unfortunately, we really cannot do this sort of thing using remote views. An alternative approach might be to use a stored procedure to do the search, build a result set on the server, and then use a remote view to query that result set. Unfortunately (again), since we cannot use remote views to call stored procedures, we still need to deal with SPT anyway. Either way, the limitations of the remote view demand that we adopt another strategy when we need this type of functionality, if at no other time. In short, for anything but the simplest of work you really do need to get to grips with SQL Pass-Through, so that's what we'll cover next.

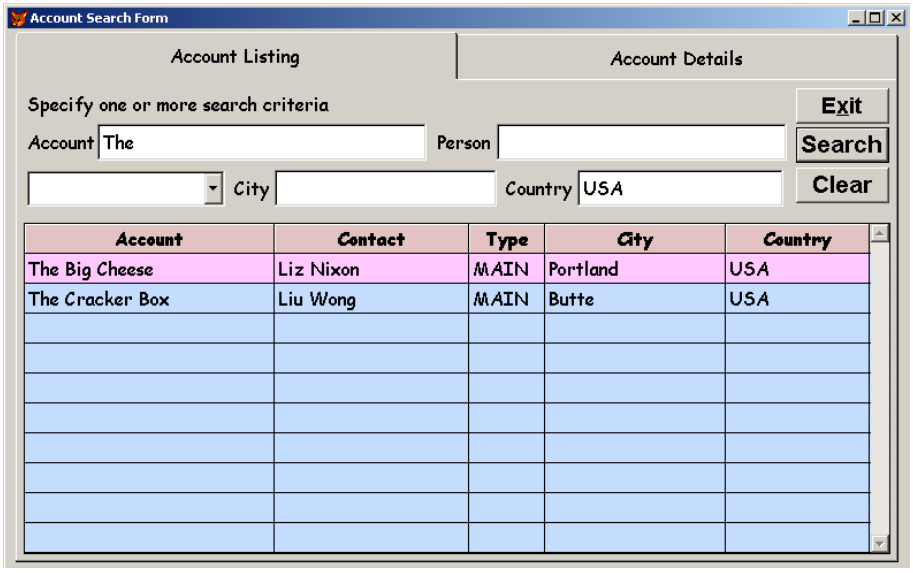


***Figure 11**. Ad-hoc search functionality that is useful in client/server applications.*

What should I use instead of remote views then?

The answer is to use SQL Pass-Through directly in code to return native Visual FoxPro cursors. SPT is the technology that allows us to send commands and instructions directly from Visual FoxPro code to a back-end server via an open ODBC connection. Using SPT we can gain greater access to the functionality of the server than is possible by simply using remote views. We have already met three of Visual FoxPro's SPT functions in the context of connections: `SQLSTRINGCONNECT()`, `SQLCONNECT`, and `SQLDISCONNECT()`. One major benefit of using SPT is that the only thing it requires is that the correct drivers and connections exist. There is no need to use a Visual FoxPro DBC, or to create Visual FoxPro connection objects. A comparison of the feature set for SPT and remote views is given in **Table 7**.

***Table 7**. Feature comparison of SPT and remote views.*

SQL Pass-Through	Remote view
Can use any statement, or command, that is supported by the server, including data definition command, administration, and security functions.	Can only use SQL Select statements.
Can access stored procedures and native functions.	No access to stored procedures or functions.
No predefinition is necessary, queries can be formulated dynamically.	Views must be predefined and cannot be altered at run time.
Query results returned as FoxPro cursors.	Query results returned as FoxPro view.
Cursors do not update the back end by default. Must set properties explicitly in order to use the `TABLEUPDATE()` function to update back end.	Persistent properties, set at design time, determine whether and how the view updates the back-end server when a `TABLEUPDATE()` is called.
Full control of Insert, Update, and Delete commands.	Control limited to the choice from pre-defined options for the Where clause.
Have no visual representation in the Visual FoxPro designers and cannot be accessed through the designers.	Has a visual representation in the DBC, a visual designer, and can be used at design time as if it were a table or local view.
Can use batched queries to return more than a single result set.	Limited to one result per query.
Properties are not persistent and no additional components are needed.	Properties, including connection information, are persistent and require a Visual FoxPro Database Container and a Visual FoxPro connection object.
Provides access to, and control over, server's transaction management.	No access to the server's transaction management.
Full support for both synchronous and asynchronous operation.	Only support synchronous operation (but can implement "background fetch").
Connection information is not embedded, can use any available connection.	Connection information is embedded with the view definition and has to be overridden explicitly if a different connection is required.

FoxPro's SPT functions

Visual FoxPro provides a comprehensive set of functions that implement SPT and that can be grouped by function (see **Table 8**). The key features of each functional group are discussed in the related subsections.

***Table 8**. Visual FoxPro's SQL Pass-Through functions.*

Function	Command	Description
Connection Management and properties	SQLCONNECT() or SQLSTRINGCONNECT()	Establish a connection to a remote server using the specified DSN or connection string. Both return a numeric *connection handle* that is required by all other functions.
	SQLDISCONNECT()	Disconnects the specified connection handle and releases the link to the server. Passing a connection handle of zero disconnects all open connections.
	SQLSETPROP() and SQLGETPROP()	These two functions allow you to retrieve and set various settings for a specified connection. Not all of the settings are always accessible for change and, in any case, caution should always be used when altering settings on an open connection.

***Table 8**. Continued.*

Function	Command	Description
Command Execution	SQLPREPARE()	Passes a statement or command to the server, which compiles it for later execution. Can dramatically speed up repeated queries, but is not supported by all servers.
	SQLEXEC()	Immediately executes the specified statement or command on the server.
	SQLMORERESULTS()	When executing asynchronously, in non-batch mode, returns the next set of results from the queue.
	SQLCANCEL()	Cancels a command sent to the server (only applicable when in asynchronous mode).
Transaction Management	SQLCOMMIT()	Commits a server-side transaction that is open on the specified connection.
	SQLROLLBACK()	Rolls back a server-side transaction that is open on the specified connection.
Miscellaneous	SQLTABLES()	Retrieves information about the data tables that are available on a given connection. Different servers support different options for this command but all should support TABLES, VIEWS, and SYSTEM TABLES.
	SQLCOLUMNS()	Retrieves information about the structure of a server table. Can report either the actual column characteristics as defined on the server, or their equivalent Visual FoxPro data types (e.g. Columns defined on the server as `VARCHAR(n)` can be reported as Visual FoxPro `CHAR(n)`).

Connection management (*Example: ConMgr.prg*)

We have already discussed, at the beginning of this chapter, the various options for establishing an ODBC connection to a back-end server. Whichever method you use, a numeric connection handle will be returned for a successful connection. This connection handle is the key to using all of the other SPT commands and it is vitally important to keep track of it—especially when you are using multiple connections.

To disconnect from a server, the `SQLDISCONNECT()` function is called with the specific connection handle that is to be terminated. Called with a parameter of 0, this function terminates all open connections. Attempting to disconnect when an asynchronous command is still executing will cause an error—you must always use `SQLCANCEL()` before disconnecting if you are running asynchronously.

We strongly advise using a connection manager object to handle the task of connecting and disconnecting from servers, and keeping track of open connections. An example of a data driven connection manager class, which uses the same tables that we defined for creating and managing DSNs (see Table 3), is included in the download code for this chapter. This class has three exposed methods for dealing with connections:

- *OpenConn()*—Expects to receive the name of a connection that is defined in the DSNCONN.DBF table. The details for establishing the connection are read from the table and the method returns a logical value indicating whether the connection was successful.

- *GetConn()*—Returns the connection handle for the first available non-busy connection. A value of zero indicates that there are no free connections, while a negative value is returned when an error occurs.
- *CloseConn()*—Expects to receive a connection handle and closes the specified connection. Note that the object's *Destroy()* method will close all open connections when the object is released, so this method is only required when you wish to close a specific connection.

Two additional methods deal with error handling:

- *GetErrors()*—Populates and returns a parameter object that has an *nErrorCount* property and an *aErrors* collection where
  - Column 1 = Error Number
  - Column 2 = Error Text
  - Column 3 = Method in which the error occurred
- *ShowErrors()*—Retrieves the same parameter object as *GetErrors()* but displays a message box with details of any errors. Intended for use at design time only.

And three further methods retrieve data dictionary information from the server:

- *GetTables()*—Expects a connection handle and, optionally, a cursor name. Returns a cursor containing the details of all tables available on the specified connection. This is a wrapper around the native **SQLTABLES()** function.
- *GetViews()*—Expects a connection handle and, optionally, a cursor name. Returns a cursor containing the details of all views available on the specified connection. This is another wrapper around the native **SQLTABLES()** function.
- *GetFields()*—Expects a connection handle, the name of a table or view on the server, and, optionally, a cursor name. Returns a cursor containing the structure of the specified entity in Visual FoxPro format (that is, field definitions reflect the Visual FoxPro data types that will be used when data is retrieved from the entity). This is a wrapper around the native **SQLCOLUMNS()** function.

Connection properties

The Visual FoxPro Help file lists all of the connection properties that can be read and set using the **SQLGETPROP()** and **SQLSETPROP()** functions. At the risk of repeating what is in the Help file, the most important Read/Write settings are listed in **Table 9**.

***Table 9**. Connection properties.*

Setting	Description
Asynchronous	Specifies whether the front-end application waits until the back-end server has completed processing. The default is False and should be used unless there are specific circumstances that require asynchronous execution, such as a very large result set. This setting is ignored by remote views, which always behave synchronously.
BatchMode	Specifies whether **`SQLEXEC()`** returns multiple result sets at once (.T.) or waits (.F.) until an explicit **`SQLMORERESULTS()`** is issued. The default is True.
ConnectTimeOut	Specifies the number of seconds of inactivity before a connection time-out error is raised. A value of 0 puts Visual FoxPro into an infinite wait state and no error is ever raised. The range is 0 to 600 and the default is 15.
DispLogin	Value determines when the ODBC Login dialog box is displayed: 1 = Display when needed (i.e. only if invalid parameters passed) 2 = Always display before connecting (avoids the need to pass parameters) 3 = Never display (generates error if login information invalid) The default is 1 but note that this *must* be set to 3 when using MTS!
DispWarnings	Specifies whether connection error messages are displayed on screen. The default is False.
IdleTimeout	Specifies, in seconds, the interval after which idle connections are terminated. A value of 0 means that the connection is never terminated. The default is 0.
PacketSize	The size of the network packet used by the connection. Adjusting this value can dramatically affect performance. The default is 4096 bytes or 4KB.
QueryTimeOut	Specifies, in seconds, the interval after which a general time-out error is raised. A value of 0 means that no time-out error is ever raised. The range is 0 to 600 and the default is 0.
Transactions	Determines how the transactions will be managed on the server. Either: 1 Automatic processing 2 Manual processing (requires explicit use of **`SQLCOMMIT()`** and **`SQLROLLBACK()`** functions.) The default is 1.
WaitTime	The interval, in milliseconds, that Visual FoxPro repeatedly checks to see whether a SQL statement has completed. The default is 100 milliseconds.

Command execution (*Example: ExecDemo.prg*)

Of the four functions that fall into this category, the main one is **`SQLEXEC()`**. The other three have limited applicability and we can deal with them very swiftly. All functions take, as their first parameter, the connection handle on which they are to operate.

- **`SQLPREPARE()`** can be used (on servers that support it) to pre-compile a SQL query and enable the server to execute it more quickly. While it can be useful if you need to run the same query several times, it is of limited value because only one query can be held in the "prepare" buffer at any time. In general, stored procedures offer a better way to handle queries that would benefit from pre-compilation.
- **`SQLCANCEL()`** is only relevant when running asynchronously. It provides the means of canceling an operation that is running on the back end and should always be used before attempting to disconnect an asynchronous connection. In practice, therefore, this function should always be used with **`SQLGETPROP( 'ASYNCHRONOUS' )`**.

- **SQLMORERESULTS()** is only relevant when the server's batch processing capability has been disabled. The default behavior is that, if passed multiple SQL commands, they will be executed immediately in the order in which they are passed and the result sets returned accordingly. This behavior is controlled, in Visual FoxPro, by the *BatchMode* property of the connection and, when set to False, SQL commands are held and only executed by issuing **SQLMORERESULTS()**. However, this is another command that, in the context of a client/server application, has little applicability since it would be most unusual to batch multiple commands in a single statement and *not* want them executed immediately.

Once a connection has been established, **SQLEXEC()** is the main function that you use for dealing with the server. This function always requires that the connection handle be specified, and can accept two additional parameters. The first is the statement that is to be executed and is mandatory unless a statement has already been sent to the server using **SQLPREPARE()**. The second, optional, parameter is the name of the Visual FoxPro cursor that will be created for the result set. By default this will be **SQLRESULT** and it is overwritten each time a **SQLEXEC()** command is run.

The function returns a numeric value indicating the result as follows:

Value	Meaning
-1	Error (either on the connection or on the server)
0	Still executing
>0	Number of result sets returned

The limitations of this function are defined by what the server will accept via ODBC. For example, with SQL Server it is possible to execute SQL commands, create SQL statements as stored procedures, and execute pre-defined stored procedures. The following code (which can be found in the download code for this chapter as EXECDEMO.PRG) shows how **SQLEXEC()** may be used.

```
*********************************************************************
* Program....: EXECDEMO.PRG
* Compiler...: Visual FoxPro 07.00.0000.9400 for Windows
* Purpose....: Illustrate use of SQLEXEC()
*********************************************************************
LOCAL lnConHandle, lnRes, lcSql, lcProc

*********************************************************************
***  Connect to SQL Server
*********************************************************************
lnConHandle = SQLCONNECT( 'SQLPubs', 'sa', 'sa' )
*** Ensure connection is synchronous
lnRes = SQLSETPROP( lnConHandle, "Asynchronous", .F. )
*** And disable ODBC Error Messages
lnRes = SQLSETPROP( lnConHandle, "DispWarnings", .F. )

*********************************************************************
*** Execute an SQL Select, return results in cursor "curResults"
*********************************************************************
lcSql = "SELECT PU.pub_name, PU.city, TI.title " ;
      + "  FROM publishers PU, titles TI " ;
```

```
      + " WHERE TI.pub_id = PU.pub_id " ;
      + " ORDER BY PU.city, PU.pub_name, TI.title "
lnRes = SQLEXEC( lnConHandle, lcSql, "curResults" )

*****************************************************************
*** Create a simple Stored Procedure
*****************************************************************
lcProc = "CREATE PROCEDURE showsales @FindID char (4) " ;
       +      "AS SELECT * FROM sales WHERE stor_id = @FindID"
lnRes = SQLEXEC( lnConHandle, lcProc )

*** Call the Procedure from VFP and return result in cursor "V_Sales"
lnRes = SQLExec( lnConHandle, "execute showsales '7066'", 'v_sales')

*** Drop the procedure
lcProc = "drop procedure [dbo].[showsales]"
lnRes = SQLEXEC( lnConHandle, lcProc )

*****************************************************************
*** Disconnect (safely) from SQL Server
*****************************************************************
lnRes = SQLCANCEL(  lnConHandle )
lnRes = SQLDISCONNECT( lnConHandle )
```

Transaction management (*Example: TxnDemo.prg*)

The default setting for the *Transactions* property of a Visual FoxPro connection is *automatic*, which means that the management of transactions is left up to the server. In this mode, transactions are immediately committed if the command succeeds, or rolled back if the command fails, when the initiating command completes on the server. This can be thought of as being equivalent to using row buffering in Visual FoxPro because each update command is treated by the server as an isolated transaction. Clearly this is not appropriate when managing multiple table updates, or batches of updates that require several **`SQLEXEC()`** calls, and our best advice to you is to control when transactions are closed at all times rather than rely on the server. Of course, if you are committed to using only remote views, and do not want to use any SPT at all, then you have no choice. As we have already noted (see Table 7), one of the limitations of remote views is that they do not allow control over server-side transactions. The default behavior for most back-end servers is to start transactions automatically upon receipt of an Insert, Update, or Delete statement (which may explain why there is no SPT *StartTransaction* function). If your server is configured to use "implicit transactions" like this, you do not need to do anything other than to set the connection's *Transactions* property to Manual (2) to gain control over how transactions are terminated. The following code connects to SQL Server and puts the connection into "manual transaction mode":

```
*****************************************************************
***  Connect to SQL Server
*****************************************************************
lnConHandle = SQLCONNECT( 'SQLPubs', 'sa', 'sa' )
*** Disable ODBC Error Messages
lnRes = SQLSETPROP( lnConHandle, "DispWarnings", .F. )
*** Set Transactions to manual
lnRes = SQLSETPROP( lnConHandle, 'Transactions', 2)
```

When set up like this, the first update command issued initiates a transaction on the server that remains open until we explicitly issue either a commit or a rollback instruction. Once a transaction has been started on a connection, any further update statements on the same connection are added to the existing transaction. Both **SQLCOMMIT()** and **SQLROLLBACK()** return values of 1 if they succeed and –1 if they fail. In the latter case we can use **AERROR()** to determine what went wrong. The following code provides basic transaction management that allows us to send multiple updates and wrap them in a single back-end transaction:

```
*********************************************************************
*** Update a table
*********************************************************************
lcSql = "INSERT INTO publishers (pub_id, pub_name, city, state ) " ;
      + "VALUES ( '9934', 'Krakins Publishing, Inc', 'Akron', 'OH' )"
lnRes = SQLEXEC( lnConHandle, lcSQL )
IF lnRes = 1
  *** Only try the second if the first succeeds. If it has failed
  *** there's no point in trying this one because we'll roll back anyway
  lcSql = "INSERT INTO publishers (pub_id, pub_name, city, state ) " ;
        + "VALUES ( '9935', 'G&G Publishing, Inc', 'Detroit', 'MI' )"
  lnRes = SQLEXEC( lnConHandle, lcSQL )
ENDIF
*********************************************************************
*** Commit if successful, Rollback if fails
*********************************************************************
IF lnRes = 1
  lnRes = SQLCOMMIT( lnConHandle )
  IF lnRes = 1
    lcStatus = "All Updates Succeeded"
  ELSE
    *** Commit Failed
    AERROR( laErr )
    *** Roll back transaction
    SQLROLLBACK( lnConHandle )
    lcStatus = laErr[2]
  ENDIF
ELSE
  *** Update Failed
  AERROR( laErr )
  *** Roll back transaction
  SQLROLLBACK( lnConHandle )
  lcStatus = laErr[2]
ENDIF
```

If we wish to assume complete control of how transactions are initiated, then we have to turn off implicit transactions on the server. In our experience most DBAs consider this a bad idea and will not normally allow it. However, it can be done and, in SQL Server, the **IMPLICIT_TRANSACTIONS** command does the job. Assuming that we have the appropriate permissions, we can use SPT to issue the necessary command from within Visual FoxPro:

```
*** Set automatic transactions off
SQLEXEC( lnConHandle, "SET IMPLICIT_TRANSACTIONS OFF" )
```

Irrespective of whether implicit transactions are enabled, issuing an explicit **BEGIN TRANSACTION** command always initiates a new transaction on the connection, allowing you to

create "nested" transactions when necessary. However, before using nested transactions you should check your server's documentation to ensure that they are supported in the way in which you expect.

For example, SQL Server allows for multiple transactions to be initiated, and provides a function (`@@TRANCOUNT`) that will return the number of open transactions on a connection. However, while the effect of a commit command is to close the last transaction (thereby reducing the transaction count by one), the impact of a rollback command is to close *all* open transactions (reducing the transaction count to zero). If you use nested transactions in Visual FoxPro so as to be able to roll back a portion of a complex update without affecting the remainder, you will need to change your strategy if you move to SQL Server. The download code for this chapter includes an example of explicit transaction management in the program named FULLTXN.PRG.

Miscellaneous

The last two SPT functions retrieve information about the database from the server. The connection manager class described earlier in this chapter includes methods that use these functions to retrieve database and table information.

The `SQLTABLES()` function accepts either `TABLES`, `VIEWS`, `SYSTEM TABLES`, or any other table type that may be valid for the data source, and returns a cursor containing information about entities that match the specified type. This cursor includes the owner, name, and type together with any remarks entered into the server's data dictionary.

The `SQLCOLUMNS()` function accepts the name of a specific server table or view and a format (either `FOXPRO` or `NATIVE`) and returns the structure in a cursor. However, note that the information in `FOXPRO` mode is limited to the absolute basics: field name, type, width, and decimal places. On the other hand, `NATIVE` mode will return whatever information is defined on the server, which is usually more extensive and will include the setting of NULL support for columns at the very least.

Should I run in synchronous or asynchronous mode?

This is one of the key questions that must be asked when designing any client/server application. In *synchronous* mode, the server retains control until it has finished processing whatever command, or set of commands, it has been given. This means that if you execute commands or queries that take a significant amount of time to complete, your application appears to "hang" because the front end cannot regain control of the PC until the back end has finished.

In *asynchronous* mode, control returns immediately to your application after calling a SPT function. However, to determine whether that function has finished, your application must keep calling the same SQL Pass-Through function until a value other than zero (which means "still executing") is returned. This does allow your front-end application to regain control so that you can display progress information while a back-end query or process is executing.

Unfortunately, there is no definitive answer; the mode you use will depend on the nature of your application and the environment in which it is to be run. You can even mix modes, switching your connection between synchronous and asynchronous mode as needed. In fact, by manipulating the connection settings for Synchronous and Batch operation you can create four distinct "modes" as described next.

Synchronous Batch mode

In this mode, control isn't returned to your application until *all* result sets have been retrieved. You can specify the name of the first cursor in the set as a parameter in the original function (Visual FoxPro will use `SQLRESULT` if no name is specified). Additional result sets will be named automatically by adding a sequential number to the name of the cursor. Thus a call that returns three result sets to a cursor named *curRes* would generate cursors named *curRes*, *curRes1*, and *curRes2*. This is the default mode for Visual FoxPro connections and is the most widely applicable mode for applications.

Synchronous Non-Batch mode

In this mode, the first statement retrieves the first result set and returns a value of 1. You then call `SQLMORERESULTS()` repeatedly. If you don't specify a new name for each new cursor, unique names are generated by taking the name of the first result set and adding a sequential number to it. When `SQLMORERESULTS()` returns a value of 2, there are no more results available.

Asynchronous Batch mode

In this mode, each repeat call of the original function returns a 0 (still executing) until all of the multiple result sets have been returned to the specified cursors. When all results have been retrieved, the return value is either the number of cursors, or a negative number indicating an error.

Asynchronous Non-Batch mode

In this mode, `SQLEXEC()` returns a value of 1 when it completes the retrieval of each result set. Your application must then call `SQLMORERESULTS()` repeatedly until a value of 2 is returned, indicating that no more results are available.

How do I work with SPT cursors?

The main practical difference between using SPT cursors and remote views is that SPT cursors do not automatically update their source table with changes; additional code is needed. There are two options—first, to explicitly set the necessary properties on the cursor, after it has been created, so that Visual FoxPro can use it to generate an update statement automatically. The alternative is to implement a set of classes that will handle the issues of dealing with a remote data source as transparently as possible.

How can I make a cursor updatable? (*Example: UpdCurs.scx*)

Cursors have a comprehensive list of properties that can be accessed using the `CURSORGETPROP()` and `CURSORSETPROP()` functions. The full list is documented in the Help file, but for convenience, the subset required to make a cursor updateable are listed in **Table 10**. Once these have been set, the native `TABLEUPDATE()` function can be used to send changes to the back-end server in exactly the same way as for a remote view.

***Table 10**. Cursor properties that control updates.*

Property	Description
Tables	Comma separated list of all tables that are to be updated. Note: To avoid compromising data integrity a cursor should only ever update one table, so although a "list" is referred to, normally there is only one.
KeyFieldList	Comma separated list of the fields that comprise the primary key in the table to be updated. This will normally be the name of surrogate PK field.
SendUpdates	Logical field that determines whether updates get sent to the server or not. Default value is False, this must be set explicitly to True.
UpdatableFieldList	Comma separated list of fields to which updates are allowed. Only fields from this list will be updated on the server. Changes made in the cursor to other fields are simply ignored.
UpdateNameList	Comma separated list of name pairs (separated by a space) that map the Visual FoxPro cursor field names to their server counterparts. For example: "au_id dbo.authors.au_id, address dbo.authors.address"
UpdateType	Numeric value that determines whether a single update statement is generated (Value =1) or separate delete and insert statements (Value –2). Default is 1 and this is generally adequate.
WhereType	Numeric value that determines how the **WHERE** clause is formulated. There are four options: [1] Only the key fields defined in the *KeyFieldList* property [2] Key fields defined in the *KeyFieldList* property and all fields defined in the *UpdatableFieldList* property [3] Key fields defined in the *KeyFieldList* property and any fields defined in the *UpdatableFieldList* property that have been modified (Default) [4] Key fields defined in the *KeyFieldList* property and a comparison of the time stamps Assuming that the database uses surrogate primary keys, this should be set to 1 instead of the default 3 to get the most efficient update statements.

The problem is, of course, that all of these properties have to be set each time a cursor is created. In practice this means *every time that a query is run* because, just as re-running a local SQL query destroys any existing cursor, so does running an SPT query. That is why in the example form, the custom *DoQuery()* method has to include all the code for re-setting the properties.

The example form (UPDCURS.SCX) shows just what is necessary and how it can be done. Only three methods in this form have any code. The *Load* event is used to create an instance of our connection manager class and open the *CH13Demo* connection, storing the handle to a form property for ease of use later. The rest of the work is done in the two custom methods, *DoQuery()* and *DoSave()*.

The first thing that *DoQuery()* does is to set up local variables for the key elements of the task. This includes the SQL statement, the cursor name, the lists of updateable fields, and what they map to in SQL Server:

```
LOCAL lcSQL, lcCursor, lcUpdFields, lcUpdNames, lnCon, lnRes
*** Get a single record we need into variables here
lcSQL = "SELECT * FROM authors WHERE au_id = '341-22-1782'"
lcCursor = "curAddress"
lcUpdFields = "au_lname, au_fname, phone, address, city, state, zip, contract"
lcUpdNames  = "au_id dbo.authors.au_id, " ;
            + "au_lname dbo.authors.au_lname, " ;
            + "au_fname dbo.authors.au_fname, " ;
```

```
            + "phone dbo.authors.phone, " ;
            + "address dbo.authors.address, " ;
            + "city dbo.authors.city, " ;
            + "state dbo.authors.state, " ;
            + "zip dbo.authors.zip, " ;
            + "contract dbo.authors.contract"
```

The actual work is done in the next part of the method, where the connection handle is retrieved, and the SPT query is run. Notice the test for the existence of the target cursor, and the subsequent call to **TABLEREVERT()**, before the query is run. This is to prevent Visual FoxPro throwing an "uncommitted changes" error should the cursor have pending changes. Since the query is going to destroy the cursor anyway, we can safely lose any such changes here. If the query succeeds (remember that "succeeds" in this context merely means that a result set was obtained; it does not necessarily mean that any records were retrieved), the resulting cursor is made updateable.

```
WITH ThisForm
  *** Get the Connection Handle for the Query
  lnCon =.nCon
  IF USED( lcCursor )
    *** Revert the cursor first, just in case!
    TABLEREVERT( .T., lcCursor )
  ENDIF  *** Run the Query
  lnRes = SQLEXEC( lnCon, lcSql, lcCursor )
  *** If successful, make the cursor updatable
  IF lnRes > 0
    *** Set Tables property (Table for INSERT, UPDATE or DELETE)
    CURSORSETPROP( "TABLES", "dbo.authors", lcCursor )
    *** Key Field(Key field for WHERE clause)
    CURSORSETPROP( "KEYFIELDLIST", "au_id", lcCursor )
    *** Updatable Field List (Fields which are updatable)
    CURSORSETPROP( "UPDATABLEFIELDLIST", lcUpdFields, lcCursor )
    *** UpdateNameList (Back-end table name for each cursor field )
    CURSORSETPROP( "UPDATENAMELIST", lcUpdNames ,lcCursor )
    *** Set Where Type and Send update flag
    CURSORSETPROP( "WHERETYPE", 1, lcCursor )
    CURSORSETPROP( "SENDUPDATES", .T., lcCursor )
    *** Also set the Buffer/Locking Mode to Table/Optimistic
    CURSORSETPROP( "Buffering", 5, lcCursor )
    *** Update the display
    .RefreshForm(.T.)
  ELSE
    *** Query failed for some reason
    MESSAGEBOX( 'SPT query failed in DoQuery()', 16, 'Whoops!' )
  ENDIF
ENDWITH
RETURN lnRes
```

The custom *DoSave()* method is extremely simple since all we need to do is to call the native **TABLEUPDATE()** function and handle the possibility that there is an update failure. Most of the code in this example is concerned with the error handling, which here is done by reporting the error and reverting changes in the cursor.

```
LOCAL llRetVal
*** Just do a simple TableUpdate on the cursor
llRetVal = TABLEUPDATE( 2, .F., 'curaddress' )
*** If it fails, check the error log
IF llRetVal
  lcTxt = "Update Succeeded"
ELSE
  *** This is all error handling!
  LOCAL lnErrs, lnCnt, lcTxt
  lcTxt = "UPDATE FAILED" + CHR(13)
  lnErrs = AERROR( laErr )
  FOR lnCnt = 1 TO lnErrs
    IF laErr[1] = 1526
      *** ODBC Error
      lcTxt = lcTxt + laErr[3] + CHR(13)
    ELSE
      *** Some other error
      lcTxt = lcTxt + laErr[2] + CHR(13)
    ENDIF
  NEXT
  lcTxt = lcTxt + CHR(13) + "Reverting changes in cursor"
  TABLEREVERT( .T. )
  ThisForm.RefreshForm()
ENDIF
MESSAGEBOX( lcTxt, 64, "Update Status" )
RETURN llRetVal
```

Clearly, while creating and maintaining this code is not much of a problem when dealing with a single cursor in a single form, it would rapidly become tedious if it had to be repeated for every cursor, every time that a query was run, anywhere in an application. The maintenance load could, to some extent, be mitigated by storing the necessary properties for cursors in a table and creating a standard procedure (or class) to set them using the cursor name as the key to retrieve the values.

However, if your requirements necessitate going that far, it is worth going the whole way and creating a set of data classes that can deal with all the issues associated with using remote databases.

What are the data classes?

The classes described in this chapter are based on a design originally conceived and developed in 1996 by Hue Holleran and Andy Kramek to meet a very specific set of operational requirements. Since that original implementation the classes have been extensively revised to meet a more generic set of design criteria as outlined here:

- Handle the issues associated with connecting to a specific data source. It should not matter whether this is local Visual FoxPro data, or a remote SQL database (for instance, SQL Server or Oracle).
- Handle the issues of having to connect, simultaneously, to more than one data source, even if they are of different types.
- Correctly format and execute SQL Pass-Through queries irrespective of the data source type.

- Manage database updates (transactions) irrespective of the data source type.
- Handle connection and database errors.

Design considerations for the data classes

First, and most obviously, we need something to handle the process of connecting to a database and managing the connection. There are really only two possibilities. First, we could create a single *smart connection handler* class to manage any number of connections, of any type. Second, we could create a single *generic* connection class and create specific subclasses for each type of connection required.

The benefit of the first approach is that we would have only one connection handler to deal with (see **Figure 12**). The drawback is that the requirements for a "connection" may differ widely depending on whether we are connecting to a Visual FoxPro database or through ODBC or OLEDB, to some other data source. Having a single "smart" object handle all of these disparate requirements is not good practice because it greatly increases the complexity and overhead associated with the class.

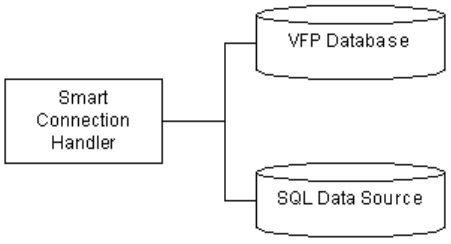


***Figure 12**. Using a single smart connection handler.*

The second approach, using specific subclasses for each type of connection, looks more promising. We can define a standard public interface for the root *connection manager* class and just subclass it as needed to handle different types of connections. However, this means that we could end up with every data-aware element in our system needing to track references to all available connections. The solution is to add a *Data Manager* object to handle the connection managers and to provide a single interface for communication with external objects (see **Figure 13**).

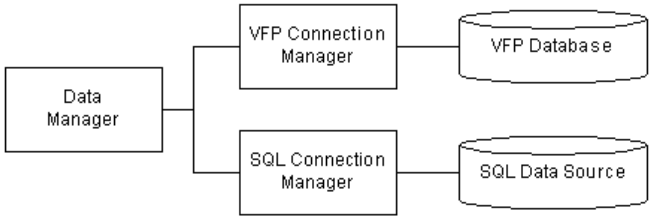


***Figure 13**. Using connection manager subclasses.*

A fortuitous benefit of this design is that it also addresses of our second key requirement because it allows us to handle multiple, simultaneous connections very easily. All that is needed is to ensure that if a required connection is not already available, it is simply added by the data manager. This is, therefore, the model that we adopted.

Our third requirement (correctly formatting SQL queries) hides a potential problem. The basic issue of formatting queries for any given data source is simple enough, but issues may arise when we consider different versions of any given data source. While it is probably true to say that the basics of SQL change little, if at all, different manufacturers do add functionality that is version-specific. For example, older versions of Oracle would not recognize the `AS` clause in a field definition, requiring instead that a local field alias be separated by a space from the field name. This changed in a recent version, so we cannot simply assume that we can treat all versions of any data source alike.

The solution is that, instead of making the connection manager responsible for the formatting, we add another object to our model to handle the specific formatting (and any other behavioral characteristics) required by different versions of a data source. The connection manager is now only responsible for ensuring that it has a reference to the appropriate behavior object when it receives notification of a query. **Figure 14** shows the revised logical design.

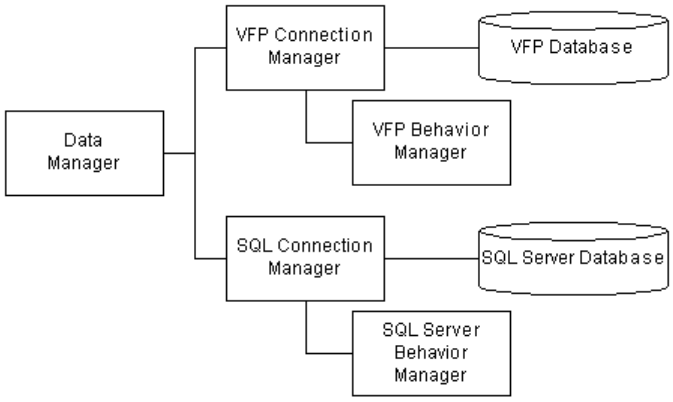


***Figure 14**. Revised logical design, incorporating behavior objects.*

The fourth and fifth requirements (transactions and error handling) are closely related because they are both concerned with reporting the status of an operation to the initiating object. Since we do not want to compromise the interface provided by the data manager, we need to provide a means for its contained objects (both Connections and Behaviors) to pass results, and error information when appropriate, back to the caller. This is achieved by having both results and errors "bubble up" to the data manager, which returns the final result, and any error information, to the caller in a parameter object. Neither of these requirements requires any further change to the logical model.

However, so far we have really only considered how to communicate with the data source. Now we need to address the issue of actually making the data available to an application. That requires two more classes.

The custom cursor class

As noted at the beginning of this chapter, one of the benefits of using ODBC to connect to a remote data source is that it returns the result of a query in a FoxPro cursor. It is important to recognize that in Visual FoxPro generally, and in this context especially, the term cursor has three different meanings.

- The word "cursor" is derived from the phrase "**CUR**rent **S**et **O**f **R**ecords" and so refers to a set of information that has been retrieved from a data store.
- The word is also used to refer to the physical entity that Visual FoxPro creates to hold the current set of records. This is actually a temporary DBF file.
- A cursor is also the object (based on the cursor base class) that Visual FoxPro creates to define the properties of the temporary table in which it stores the current set of records.

In other words, Visual FoxPro creates a cursor object that defines the properties of the cursor table into which the cursor records are placed. We are particularly interested, for the moment, in the cursor object because it provides the solution to the problem of communicating between an application and the back-end data source. However, the native base class does not have the necessary properties, and so we need to create a custom class that does two things:

First, it defines, to the data manager and the connection manager, the data source with which it wishes to communicate. This is achieved by using two properties—one to define the type of connection, the other to define the type of behavior, that the cursor requires.

Second, it must contain all of the information necessary to allow the behavior manager to correctly formulate the SQL required to execute a query or update statement. The full set of properties that we use is listed in **Table 11**. You may, of course, extend these properties if you need to but we have found that this set provides for the basic functionality.

***Table 11**. Cursor class properties.*

Property	Type	Default	Usage
cBehaviour	Char	""	Type of behavior the cursor requires.
cConnName	Char	""	Type of connection the cursor requires.
cSelFields	Char	"*"	Comma separated list of fields to be returned in a query. Default is ALL, used to build the **SELECT** clause.
cTables	Char	""	Comma separated list of tables participating in a query. Used to build a **FROM** clause when not using the **JOIN** syntax.
cPermWhere	Char	""	**JOIN** conditions to be applied every time the query is run. Used to build a **WHERE** clause when not using the **JOIN** syntax.
cPermJoins	Char	""	**JOIN** clause to use every time the query is run. Used instead of the *cTable* and *cPermWhere* if the data source supports this syntax.
cPermHaving	Char	""	**HAVING** clause to be appended in every query.
cDefOrder	Char	""	**ORDER BY** clause to be appended in every query.
cDefGroupBy	Char	""	**GROUP BY** clause to be appended in every query.

***Table 11**. Continued.*

Property	Type	Default	Usage
lUpdatable	Logical	.T.	Flag whether the cursor should update its source table.
cUpdTable	Char	""	Name of the table to be updated.
cKeyField	Char	""	Name of the key field to use in update statements.
cUpdFields	Char	""	List of fields that may be updated. If left empty, all fields are assumed to be updateable.
cLastWhere	Char	"0=1"	The last **WHERE** clause applied to the query, used to provide re-query functionality.
lUseTblBuff	Logical	.T.	Flag whether table buffering should be used on the cursor.
lPopulate	Logical	.F.	Flag whether all data should be retrieved when opening the cursor. Default behavior is not to retrieve data.

As you can see, the properties fall into four groups; the first pair defines the requirements of the cursor for connection and behavior managers. The sample code includes pre-defined subclasses set up for both Visual FoxPro and SQL Server. The last group of properties is only used internally by the data classes to define how a cursor should be opened, buffered, and re-queried.

Seven properties define how to build the query that creates the cursor. The code allows for using either the standard SQL (**SELECT**…**FROM**…**WHERE**) or the ANSI92 (**SELECT**…**FROM**…**JOIN**…**ON**) syntax, because not all servers support the latter format. Note that the code is constructed so that if the join condition is populated, both the *cTables* and *cPermWhere* properties are ignored. This means that if you use the join syntax and also wish to apply a static filter, the filter must be explicitly included in the *cPermJoins* property definition.

A group of four properties is used to define how changes made in the cursor are sent to the data source. There are some implicit assumptions here. The first is that even though a cursor may be defined as the result of joins, it may only ever update one table on the data source. This may sound restrictive, but in practice, in more than five years of working with this model, we have never found a situation where it was necessary to update multiple tables from a single cursor anyway.

The next assumption is that the key field is never updateable and that the responsibility for populating the key field is left entirely to the data source. Implicit in this is another assumption that the primary key will always be an integer (that is, a surrogate key) generated by the database. The reason for this is twofold. We need some means of identifying whether a record in the cursor is a new entry (and so requires an **INSERT**) or an update to an existing record (either an **UPDATE** or a **DELETE**). We use the presence or absence of the primary key to make this decision. Besides this, there are as many different mechanisms for generating key field values as there are data sources. So rather than try and cope with all the possible mechanisms for generating keys, we have opted for leaving it up to the server. If you are only planning to work with one data source, you may wish to change this accordingly.

Finally, note that no provision is made for mapping local field names to server-side column names in updateable cursors. The assumption is that updateable column names are never changed in the front end. (Note that long column names are not a problem because, unlike tables, cursors allow the use of long field names even though they are not bound to a database container.) This has an important benefit. By adhering to this rule, we completely separate the front-end application from any specific back end. Since we can be sure that the

field names will always be the same, it allows us to create a Visual FoxPro database that mimics the structure of, say, an Oracle database. We can then use the Visual FoxPro model to build and test our application and, when ready to "upsize" to the real back end, all we need to do is change the cursor class that we use. Since no other code changes, we can be certain that nothing will break.

The cursor class has no custom methods, and the only code is some simple manipulation in the *Init()* and *Destroy()* methods to ensure that each cursor object has a name when it is created, and that no changes are left pending when it is released.

Defining cursors (*Example: CurDefs.scx*)

The sample code included with this chapter uses a metadata table (CURDEFS.DBF) to store cursor definitions. This table has a field defined for each of the custom properties of the cursor object, plus an additional field that is used only for recording the name of a Visual FoxPro database. This value is used to populate the cursor's native *Database* property when the cursor object is instantiated. To simplify the management of this table we use a form (see **Figure 15**).

The figure shows the definition for a cursor that simply retrieves, and allows updates to all fields, in a table named *Authors*. We can assume that this table resides on a remote server because the cursor is shown here as using the *xCurSQL* class.

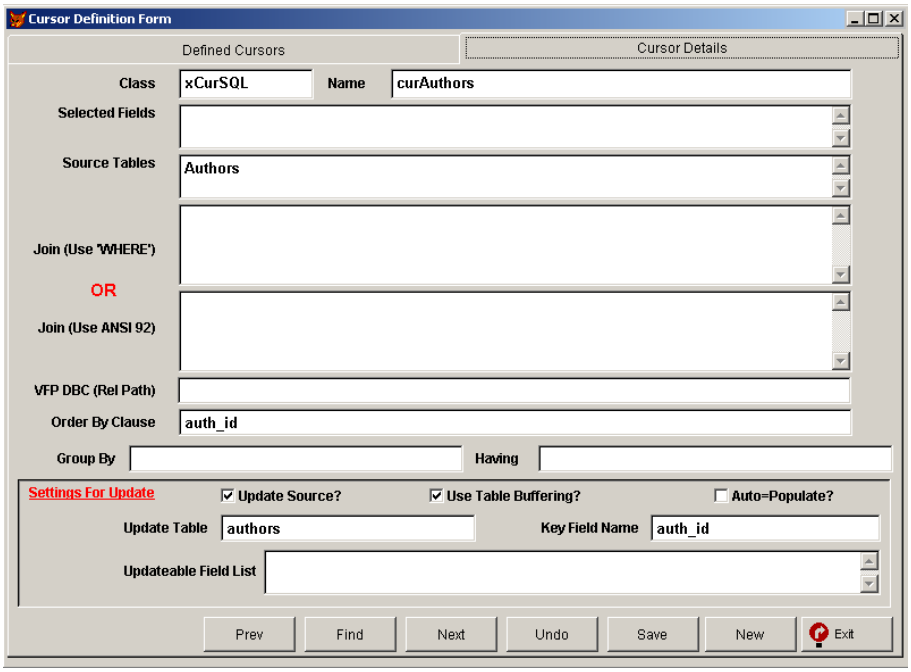


***Figure 15**. Cursor definition management form (curdef.scx).*

Managing cursors, the dataset class

The final class that we need is the *Dataset*, which has two functions. First, we use it instead of the native Visual FoxPro dataenvironment as the repository for cursors in a form. Second, it provides all the required functionality for creating and working with cursors to the form. It is, therefore, the communication link between the application and the data manager.

The *DataSet* root class defines the basic functionality and, within an application, a new subclass is created for each different set of data required. The convention that we use is that each subclass is defined as a standalone PRG file so that, in a team environment, developers can work on their own datasets without trampling over each other. However, that is merely a convention and you could use a single PRG file to contain all your dataset definitions.

Each dataset is defined in a set of three metadata tables (DATASETS.DBF, DSETLINK.DBF, and CURNAMES.DBF) that associate cursors with a dataset class. The structure of these tables is shown in **Table 12**.

***Table 12**. Dataset metadata tables.*

Field Name	Type	Usage
Datasets.dbf		
dSetPK	I (4)	Primary key
dSetType	C (5)	Identifier for grouping datasets by type (optional)
dSetName	C (30)	Dataset name
dSetDesc	M (4)	Description
dSetClass	C (20)	Name of the class to instantiate
DSetLink.dbf		
dSetCurPK	I (4)	Primary key
dSetFK	I (4)	Foreign key to DATASETS.DBF
curFK	I (4)	Foreign key to CURNAMES.DBF
CurNames.DBF		
curPK	I (4)	Primary key
DefDK	I (4)	Foreign key to cursor definition table
curType	C (10)	Cursor type either: "TABLE"—based on a single table "CUR"—based on a multi-table join
curLocn	C (50)	Relative path and file name for the cursor definition table
curName	C (30)	Name of the cursor to create
curDesc	M (4)	Description and developer notes about this cursor

Figure 16 shows how these tables are used to define a dataset named *SQAccounts*. When a dataset is instantiated, it uses the metadata described previously to determine what cursors it requires, and where to find the definitions. A cursor object is instantiated and added to the data set for each defined cursor. A hook method (*SetUpObject()*) is called from the root class *Init()* to allow for any specific setup code that may be needed in a dataset subclass.

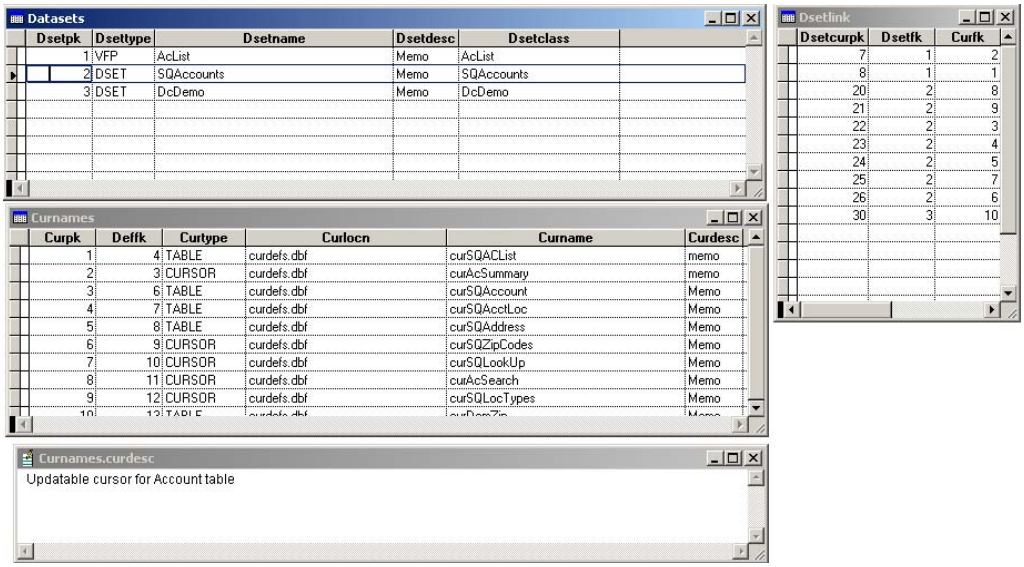


***Figure 16**. Metatdata tables for dataset definitions.*

The native *Init()* method expects to receive a reference to the data manager that the dataset is to use. It simply calls the custom *InitObj()* method, passing on the reference. (Note: If no reference is passed, the *InitObj()* method tries to instantiate the data manager using a connection named *Default*. If this fails, the dataset will not initialize and returns False.) The dataset sets its internal property (*oDM*) to point to the data manager and follows the initialization sequence as follows:

- First, *SetUpRefTables()* ensures that a copy of the metadata tables are opened in the current datasession (uses *OpenLocalTable()*).
- Next, *GetObjDetails()* gets the list of cursors for the data set and then loops through it instantiating the cursor objects and setting the properties for each of them (uses *SetupCursor()* and *AddCursor()*).
- Finally, *SetUpObject()* is called (with any input parameters) to execute the custom initialization code. Typically this will include code to build indexes, query lookup tables, and set up relations between cursors.

There are only three functional methods exposed in the interface of the dataset class.

Do()

This is the principal method that is called directly in code. It accepts up to five parameters as follows:

- The function required (the sample code supports Query, Requery, Lookup, Fetch, and GetLastID).
- The name of the cursor to which the function is to be applied (*Query()* and *Requery()*) or that is to be used as the source of the table to use on the server (*LookUp()* and *Fetch()*).
- Up to three additional parameters depending on the function as follows:
    - *Query()* accepts an ad-hoc where clause that will be appended to any permanent joins or filters defined for the cursor. *Query()* is the basic method for populating cursors.
    - *LookUp()* accepts a value to find the name of the field to search and the name of the field whose value is to be returned. It uses SQL to mimic the function of the native Visual FoxPro function of the same name and returns a single value.
    - *FetchVal()* accepts an ad-hoc where clause and the name of a field whose value is to be returned. *FetchVal()* is similar in function to *LookUp()*, also returning a single value. However, it applies the specified where clause rather than searching within a single field.
    - *Requery()* uses the saved *cLastWhere* property to re-run the last-executed query for a given cursor.
    - *GetLastID()* returns the last ID to be issued. This will mean different things to different servers. (For example, in this implementation for Visual FoxPro it will be the highest key value found in the specified table, but for SQL Server it will be the last ID issued on the current connection.)

TryUpdate()

This method sends changes made in the local cursors to their data source. The method expects to receive a comma separated list of the names of the cursors that it should try to update. The method wraps a single FoxPro transaction around the update attempt and passes each cursor in turn to the data manager, which has all the code needed to determine what type of update is required. Remember also that the dataset maintains an internal collection of cursor names and, if nothing is specified for *TryUpdate()*, will loop through its collection and try to update any cursor that is flagged as updateable (that is, where `lUpdatable = .T.`).

As the updates for each cursor are processed, a `TABLEUPDATE()` is called to remove pending changes. If all cursors are updated successfully, the Visual FoxPro transaction is committed so that the buffers for all cursors are clean. Otherwise, the Visual FoxPro transaction is rolled back, leaving the cursors with pending changes.

The method returns a simple True or False indicating whether the specified update succeeded. *TryUpdate()* has three hook methods associated with it to which code may be added if required. These are named *BeforeSave()*, *AfterGoodSave()*, and *AfterFailSave()*. Their intent, and when they are called, is obvious.

ValToStr()

This method returns a Visual FoxPro value formatted as a character string. The reason for its existence is that when sending data to other databases, the formats in which certain data types are stored differ. *ValToStr()* is called with a minimum of two, and a maximum of three parameters as follows:

- A cursor name. The specified cursor is used to determine which particular format a value should be returned in. The default, if no name is passed, is for SQL Server.
- The value to convert.
- Optionally, the data type for the return value. This overrides the usual behavior that uses the `TYPE()` function to determine the data type directly from the value. Normally this is only used when data has been transformed into something other than its original data type (usually to text) to enable us to restore the correct data type.

The class has two other exposed methods that are concerned solely with error handling (*GetErrors()* and *ShowErrors()*).

ShowErrors()

This method, as the name implies, displays the contents of the dataset's error collection on screen and, optionally, will write the errors to a log file. It is intended only for use during development and debugging and should not really be used in an application because it uses `MESSAGEBOX()` to display the errors.

GetErrors()

This method interrogates the dataset's error collection and returns the information in a parameter object that has two properties: the number of errors found (*nErrorCount*), and a three-column array (*aErrors*) containing the error number, message, and method in which it occurred. In an application, this method allows a user interface to retrieve errors from the middle tier after a failed operation. Calling *GetErrors()* also clears the dataset's error array.

The data class implementation model

Figure 17 illustrates how the various components fit together in a Visual FoxPro application. Notice that the data manager is instantiated as a "global" object. Our personal preference is to add a property to the Visual FoxPro _Screen Object (this object still exists even if the FoxPro desktop is not visible) and to use that to hold the reference to the data manager.

This means that only one data manager is required for each instance of the application, which also means that we only need one connection per instance to the data source. Forms simply get a reference to the data manager when they are instantiated, and pass that reference to their contained datasets.

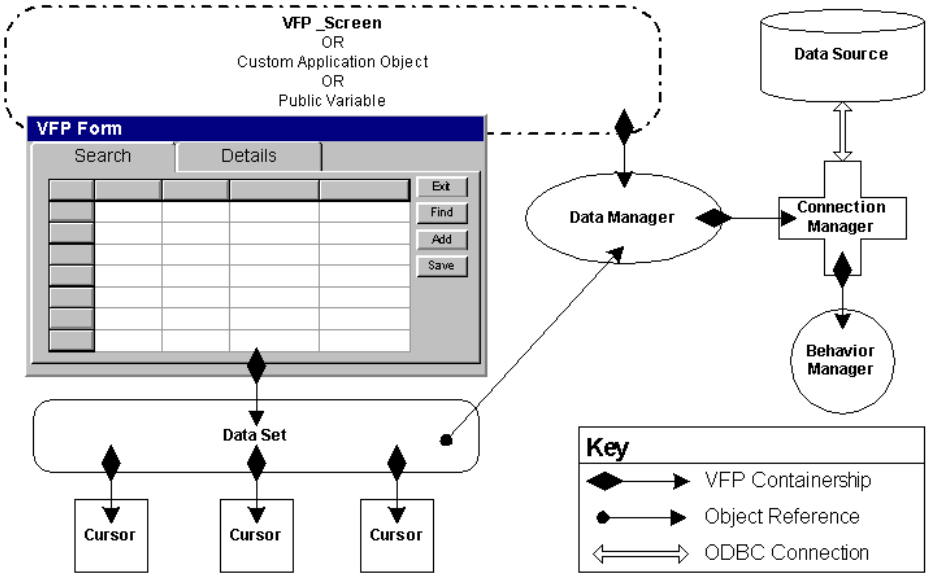


***Figure 17**. Implementing the data classes.*

How do I use the data classes? (*Example: Dataclass Directories*)

The sample code for this chapter includes a subdirectory named *DataClass* that contains a full working set of classes that will run against either Visual FoxPro or SQL Server (or both!). There is far too much code involved in these classes to work through every line in the text, so we have put together a sample form (shown in the section "The example form") that illustrates how the data classes are used in an application environment. The following sections walk through the process of setting up and using the various elements that go into making this form work.

Setting up the example databases

Unfortunately, the design of the standard SQL Server Pubs database that we have been using to illustrate this chapter does not conform to the standards that would enable us to use it with the data classes. The reason is that primary keys have not been defined for all tables, and even those that do have PKs, do not use surrogate keys. So in order to illustrate the capabilities of the data classes we have included a Visual FoxPro Database named *DemoData* and a SQL Server script (DEMODATA.SQL) that will create the same database and table structures in SQL Server and will add a new user (named *DemoData*). The file DEMODATA.BUP contains a backup of the database that can be used to populate the tables by using SQL Server's `RESTORE DATABASE` function. The Visual FoxPro database and the SQL Server database are exactly the same, and you may use either database to run the form. To switch from one to the other simply set the cursor class property for the cursors (defined in the CURDEFS.DBF table) to either *xCurVFP* or *xCurSQL* as appropriate. No other changes are needed!

A DSN will also be needed to connect to the SQL database, and a default setup has been defined in the metadata table CONNECTS.DBF. If these settings are not acceptable, you will need to change this data to reflect your requirements.

The sample code unzips into a directory named *DataClass* that has two subdirectories, named *Data* (which contains the Visual FoxPro version of the sample database) and *MetaData* (which is where the data class metadata database is stored). **Table 13** gives the full list of files and their locations.

***Table 13**. DataClass sample files.*

Location	File name	Comment
\DataClass	DATACLASS.PJX & pjt	Dataclass Project.
	ROOTCLASS.VCX & vct BASECTRL.VCX GENFORMS.VCX & vct	Visual class libraries used in the example forms.
	CURDEF.SCX & sct	Cursor definitions and maintenance form.
	DSETBASE.PRG	Dataset class source code. Definitions for a subclass of the Custom base class, our "base" object and the data set.
	DATACLASS.PRG	The Data class source code. Definitions for data manager, behavior managers, and connection managers.
	SETUPDCL.PRG	Environment setup program to run the data classes.
	SQACCOUNTS.PRG	Dataset definition for the Accounts form.
	FRMACCTS.SCX & sct	Example Accounts form.
\DataClass\Metadata	DATACLAS.DBC, dct, & dcx	Metadata database.
	CONNECTS.DBF & cdx	Connection settings (configuration for ODBC DSN and Visual FoxPro database).
	CONPROPS.DBF, fpt, & cdx	Connection settings, used to set properties with **`SQLSETPROP()`**.
	CURDEFS.DBF, fpt, & cdx	Cursor definitions.
	DATASETS.DBF, fpt, & cdx	Dataset header table (names of datasets).
	DSETLINK.DBF & cdx	Dataset link table (maps cursors to datasets).
	CURNAMES.DBF, fpt, & cdx	Cursor usage in datasets (Hold FK to Curdefs).
	DEMODATA.SQL	SQL Server 2000 script to create the DemoData database. Run from Query Analyzer.
	DEMODATA.BUP	Backup of SQL Server 2000 DemoData database. Can be restored (with force option) to re-create the data using Enterprise Manager.
	NEXTID.DBF & cdx	PK generation table.
	ERRORMSG.DBF & cdx	Error message table used by the data classes.
\DataClass\Data	DEMODATA.DBC, dct, & dcx	Example database.
	ACCOUNT.DBF, fpt, & cdx	Account header table (Name and Acct number).
	ACCTLOC.DBF & cdx	Account location link table (Contact information).
	ADDRESS.DBF & cdx	Addresses table.
	LUTYPES.DBF & cdx	LookUp header table.
	LUVALUES.DBF & cdx	LookUp items table.
	NEXTID.DBF & cdx	PK generation table.
	ZIPCODES.DBF & cdx	Extract of US ZIP code information (not used in this example).

The example form (*Example: frmAccts.scx*)
The example form illustrates a typical two-page form used in our client/server applications. The first page provides the user with the ability to carry out a search of the database using any combination of several possible criteria. The results are displayed in a read-only grid (see **Figure 18**).

The second page retrieves the details for a specific customer and provides simple Add, Edit, and Delete functionality (see **Figure 19**).

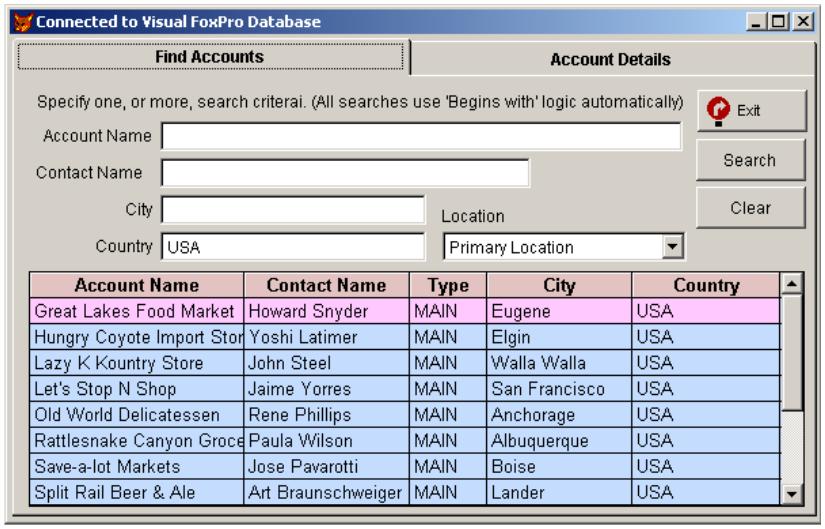


***Figure 18**. Search page of the Accounts form (frmaccts.scx).*

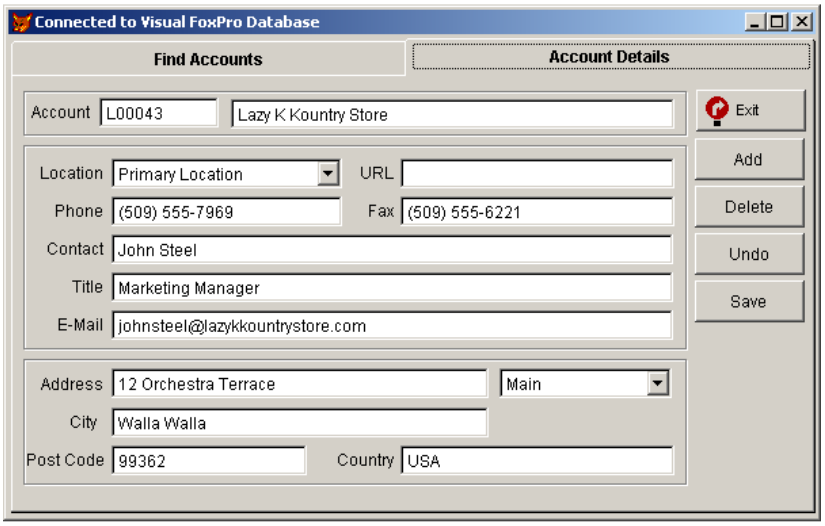


***Figure 19**. Details page of the Accounts form (frmaccts.scx).*

Cursor definitions (*Example: Curdefs.dbf*)
In order to support the example form, a total of six cursors are required as shown in **Table 14**.

***Table 14**. Cursors required for the Accounts example form.*

Cursor	Tables	Comment
curSQAccount	Account	Simple view (**SELECT ***) of the Account table. Used for updating the source table.
curSQAcctLoc	AcctLoc	Simple view (**SELECT ***) of the AcctLoc table. Used for updating the source table.
curSQAddress	Address	Simple view (**SELECT ***) of the Address table. Used for updating the source table.
curAcSearch	Account, AcctLoc, Address, Luvalues	Multi-table join, used to populate the read-only results grid on the first page of the form.
curSQLookUp	Lutypes, Luvalues	Cursor for handling look-ups (i.e. for decoding the stored FK).
curSQLocTypes	Lutypes, Luvalues	Extract from the look-up tables of the list of LocationTypes. Used to populate the Location combo box on page two of the form.

As you can see, we require a separate cursor for each table that is to be updated (this was, you will recall, one of the underlying rules of the data classes: *One cursor can only ever update one table*). One additional cursor is required to populate the grid and one as the record source for the Locations combo box. The last is a static definition for a cursor that can be used to look up a key into the look-up details table and retrieve the associated description. All of the cursors are defined in CURDEFS.DBF and the details can be inspected using the CURDEF.SCX form.

Dataset definition (*Example: Datasets.dbf, DSetLink.dbf and CurNames.dbf*)
The form will only require one dataset, which is named *SQAccounts*. The dataset tables have been set up to associate the six cursors defined in CURDEFS.DBF with this dataset name. A new subclass of the dataset base class has been created for this dataset in the file SQACCOUNTS.PRG. All that is required in a subclass definition is to handle any post-initialization code that we may need in the *SetUpObject()* method. In this case, all that we do is index the search result cursor after it has been initialized so that clicking on the column header in the grid sets that column as the sort order (that functionality is defined in the grid class). Here is the subclass definition:

```
DEFINE CLASS SQAccounts AS xDataSet
  *** This is the hook method called after all cursors
  *** have been initialized
  FUNCTION SetUpObject( tuCondition)
    *** Index the search cursor
    SELECT curACSearch
    INDEX ON caddcntry  TAG caddcntry
    INDEX ON caddcity   TAG caddcity
    INDEX ON cluvcode   TAG cluvcode
    INDEX ON clocperson TAG clocperson
    INDEX ON cAcName    TAG cAcName
  ENDFUNC
ENDDEFINE
```

Of course, there is no reason at all why additional methods should not be added to dataset subclass definitions to handle specific functionality that is not already provided for in the data classes.

Setting up the environment

In order to use the data classes the necessary environmental setting must be made. We use a startup program (SETUPDCL.PRG) to set paths, open the class libraries, instantiate the data manager, and connect to the databases. Note that for this example we are adding a property to the Visual FoxPro *_Screen* object to provide a global reference for the data manager. If you prefer to use an application object (or even a public variable), that is just fine; amend the startup program accordingly.

The code is pretty well self-explanatory and begins by setting up the path, forcing **ASSERTS** on, and setting a global variable (*glCopySQL*), which ensures that any SQL statement that is executed by the data classes is echoed to the Windows Clipboard. This is a useful feature to remember when trying to debug statements that do not appear to work!

```
**********************************************************************
* Program....: SETUPDCL.PRG
* Compiler...: Visual FoxPro 07.00.0000.9262 for Windows
* Purpose....: Data Class Environment Setup Program
**********************************************************************
CLOSE DATABASES ALL
CLEAR
LOCAL lcPath
************************************************
*** Start with the Path
************************************************
*** This is Dev Mode - Full Path required
lcPath = (HOME() + ";" ) + (FULLPATH( CURDIR() ) + ";METADATA;DATA")
SET PATH TO (lcPath)
*** Note there are several ASSERT statements in the Data Classes,
*** especially when things go wrong (like updates failing) because
*** they give you the opportunity to go into DEBUG mode right there.
*** If you want to disable them, just SET ASSERTS OFF instead
SET ASSERTS ON
*** And set the variable which makes data classes copy SQL to clipboard
RELEASE glCopySQL
PUBLIC glCopySQL
glCopySQL = .T.
```

Next we open the visual class libraries and procedure files, and create the property to hold the reference to the data manager.

```
************************************************
*** Now we should be able to open the class libraries without errors
************************************************
SET CLASSLIB TO rootclass, genforms, basectrl ADDITIVE
************************************************
*** And the Procedure files
************************************************
SET PROCEDURE TO dataclass, dsetbase ADDITIVE
************************************************
```

```
*** We need to add the Data Manager and Connect to SQL Server
*** So make sure we have the _Screen property
************************************************
IF NOT PEMSTATUS( _Screen, 'oDM', 5 )
    _Screen.AddProperty( 'oDM', NULL )
ENDIF
```

Here is where we create the data manager and, if the object instantiates, try and set up the connection to the Visual FoxPro database. If that succeeds we then set up the ODBC connection to SQL Server. The connection parameters for both connections are defined in CONNECTS.DBF, while the connection properties to use for the ODBC connection are defined in CONPROPS.DBF.

```
************************************************
*** Add the Data Manager and Connect. Check for success
************************************************
_Screen.oDM = CREATEOBJECT( 'xDatMgr', 'VFPDClas' )
IF VARTYPE( _Screen.oDM ) = "O"
    *** We have the Data Manager, but did it connect?
    lnCon = _Screen.oDm.GetHandle( 'VFP' )
    IF lnCon < 1
        MESSAGEBOX( 'Cannot Connect using VFPDClas', 16, 'Setup Error' )
        RETURN
    ENDIF

    *** OK, now add the SQL Connection
    _Screen.oDM.AddConnection( 'SQLDClas' )
    lnCon = _Screen.oDm.GetHandle( 'SQL' )
    IF lnCon < 1
        MESSAGEBOX( 'Cannot Connect using SQLDClas', 16, 'Setup Error' )
        RETURN
    ENDIF
ELSE
    MESSAGEBOX( 'Cannot create the Data Manager', 16, 'Fatal Setup Error' )
    RETURN
ENDIF
```

We are establishing both connections so that we will later be able to switch the form from running against Visual FoxPro to running against SQL Server, but in practice you would only create the connection that you actually wanted to use. Having said that, these classes were originally designed and implemented to support an application that drew data simultaneously from two separate back-end databases so we actually did have multiple simultaneous connections.

(Note that you can simulate that environment by having this form query from one source, but update another. All that is needed is to set the *xCurClass* field in CURDEFS.DBF so that the updateable cursors use one cursor class while the query cursors use the other. Of course, that does require ensuring that the keys to data in both databases are identical.)

Running the form (*Example: frmAccts.scx*)

Having set up the environment we can simply run the form using the normal Visual FoxPro syntax:

```
DO FORM FrmAccts
```

When the form was built, all the bound controls (the grid on the first page and the textboxes and combos on the second page) were set up with the appropriate cursor and field names entered into their *ControlSource* properties. In order to avoid errors when the form is instantiated, we must ensure that the cursors are created before the Form's *Init()* fires, which means in the *Load()* event's method code.

The Load() method

The root class form, on which the class we are using (*GenForms::xFrmStdData*) is based, defines a private datasession for the form and has some code in its *Load()* to set the environment. So the first thing we must do is to call **DODEFAULT()** to ensure those settings are respected. You can inspect (and modify) them in the *xFrm* class, which is found in the ROOTCLAS.VCX class library. Next we check for the presence of the data manager; if it isn't already there, we simply abort the form's initialization by returning False. (Although not documented, this behavior has been in Visual FoxPro since Version 3.0, and returning False from the *Load()* stops the form from instantiating.)

```
DODEFAULT()
*** If we don't have a data manager - abort
IF NOT PEMSTATUS( _Screen, "oDM", 5 )
  MESSAGEBOX( "Data Manager must be created before calling this form", ;
              16, "Cannot load Form" )
  RETURN .F.
ENDIF
*** Set Data Manager and Reference
WITH This
    .oDm = _Screen.oDM
    .oDs = NEWOBJECT( 'SQAccounts', 'SQAccounts.prg', NULL, This.oDm )
ENDWITH
```

If all is well, the form sets its own *oDM* property to point to the data manager and instantiates the dataset, passing a reference to its *oDM* property as a parameter. (Note that the dataset will actually accept *two* parameters. The first is mandatory but the second is optional. Any second parameter will be passed through to the *SetUpObject()* method and so must be handled explicitly in the subclass definition.)

That is all the code there is. As described previously (see the section "Managing cursors, the dataset class" earlier in this chapter for details), the data set reads the metadata to determine which cursors to instantiate. Cursors are opened and populated using a default filter of **WHERE 0=1**, which is the SQL equivalent of saying **WHERE .F.** so all that is created is an empty structure. This allows controls to be bound to the cursor even though we do not yet know what data to retrieve.

To populate a cursor with all available data on initialization, just set its *lPopulate* property to True. Note that auto-populating cursors should really only be done with local look-up tables, or other small sets of data, because of the possibility of inadvertently bringing down large volumes of data with concomitant deleterious effects on performance.

The Init() method

The *Init()* method merely initializes the custom property (*CurKeyVal*) that is used to track the PK of the currently selected record in the search cursor and sets the form caption. Note that the code is reading the *cConnName* property of the search cursor *object* to determine whether a Visual FoxPro or SQL Server class is used for this cursor. The caption is then set accordingly.

```
LOCAL lcRunning
DODEFAULT()
*** Initialize custom properties
WITH ThisForm
  *** Current PK in the search Cursor
  .curKeyVal = 0
  *** And set the caption
  lcRunning = SUBSTR( This.oDS.curAcSearch.cConnName, 4 )
  lcRunning = IIF( lcRunning = "SQL", "SQL Server", "Visual FoxPro" )
  .Caption = "Connected to " + lcRunning  + " Database"
ENDWITH
```

This highlights the fact that each cursor has an object, of the same name, which is contained by the form's dataset. All of the properties of a cursor are always available, and can be changed, by accessing this object through the form's reference to its data set (*oDS*).

You may be thinking that this means that you can change the definition of a cursor at run time, and you are right, you can! However, if a physical cursor with the required name already exists when a command is executed by the data classes, it is not actually closed. Instead, the query is run to a temporary cursor, the original is cleared, and then the new data is appended. This avoids all the problems that can arise when a cursor is closed (like grids losing their marbles and going blank), but it does mean that if you change the cursor definition, the append will fail unless you first close any existing cursor manually. This is the only situation in which you need to worry about controlling the cursor directly, and in our experience it is not something we do every day.

"Find Accounts" page Activate() and Deactivate()

The code in the search page is very simple indeed. Upon activation the page calls our custom *RefreshForm()* method. This method (which is defined in *RootClass::xFrm*) wraps a call to the standard *Refresh()* method to include setting *LockScreen* and calls to *BeforeRefresh()* and *AfterRefresh()* hooks.

```
ThisForm.RefreshForm()
```

Upon deactivation of the page, the primary key of the currently selected record in the search grid (zero if no records are in the cursor) is stored to the form's *curKeyVal* property.

```
ThisForm.curkeyval = IIF( RECCOUNT('curAcSearch') > 0, curAcSearch.iacpk, 0 )
```

Implementing the Search functionality

The Search button is an instance of our generic command button class (which was described in Chapter 1), and it merely calls the form's custom *Search()* method when it is clicked. This method builds a where clause that will be applied to the search cursor by retrieving the contents of the un-bound textboxes:

```
WITH ThisForm.pgfaccts.Page1
  *** Get whatever is specified by the user into variables
  lcAcct  = ALLTRIM( .txtAcct.Value ) + "%"
  lcPsn   = ALLTRIM( .txtPerson.Value ) + "%"
  lnType  = INT( VAL( .cboType.Value ))
  lcCity  = ALLTRIM( .txtCity.Value ) + "%"
  lcCntry = ALLTRIM( .txtCntry.Value ) + "%"
ENDWITH
```

The same logic is used for each variable as follows. Note that we are adding a "*%*" qualifier to each value to ensure that partial matching is used in searches (which Visual FoxPro does by default, but SQL Server does not). The actual mechanism can, of course, be anything that you want it to be, and one obvious refinement would be to add a checkbox to the form to toggle exact matching on and off.

```
*** Now build the Where clause (the same logic is repeated for each variable)
IF ! EMPTY( CHRTRAN( lcAcct, '%', '' ))
  lcWhere = lcWhere + IIF( !EMPTY( lcWhere ), " AND ", "" )
  lcWhere = lcWhere + "RTRIM( cacname ) LIKE " ;
          + This.oDs.ValToStr( 'curAcSearch', lcAcct )
ENDIF
```

As a matter of courtesy we like to warn users that, if they do not specify anything, they will get all data. This is particularly important if your back-end data source contains millions of records.

```
*** Apply the Filter (if we got one!)
IF EMPTY( lcWhere )
    IF MESSAGEBOX( 'Do you really want to retrieve ALL data?', ;
                   36, 'No Search Criteria' ) = 7  && No, do not do this
        RETURN
    ENDIF
ENDIF
```

Finally, we can run the query. The syntax here, as always, is the standard data class *Do()* syntax and follows the sequence: *Function Required*, *Cursor Name*, *Additional Parameters*. In this case we want to *Query* the *curAcSearch* cursor using the filter contained in the *lcWhere* variable and then update the display, thus:

```
This.oDS.Do( 'Query', 'curAcSearch', lcwhere )
ThisForm.RefreshForm()
```

The Clear button merely calls the form's *ResetSearch()* method to re-initialize the text boxes. Most of the real action occurs on the details page of the form.

"Account Details" page Activate() and Deactivate()

The Account Details page is using three updateable cursors, and so upon activation of the page we need to query all three. You will have noted that the search cursor was defined to include the PK for each table in every record, and since we updated the form's *CurKeyVal* property in the *Deactivate()* of the first page, we have a simple mechanism for doing this. However, to avoid unnecessary calls to the server, we only want to do it if the user has actually changed records, and is not in *Add* mode. The code in the *Activate()* method handles all this:

```
WITH ThisForm
  *** If we are in Add mode - just pass on through
  IF .cMode = "A"
    RETURN .T.
  ENDIF
  *** Only Re-Query if needed
  IF RECCOUNT( "curAcSearch" ) > 0 AND curSQAccount.iacpk = ThisForm.curkeyval
    *** We don't need to do anything just now
  ELSE
    *** Need to re-query the cursors
    lcCurKey = TRANSFORM( ThisForm.curkeyval )
    lcWhere = 'iacpk = ' + lcCurKey
    .oDs.Do( 'Query', 'curSQAccount', lcWhere )
    lcWhere = 'iacfk = ' + lcCurKey
    .oDs.Do( 'Query', 'curSQAcctLoc', lcWhere )
    lcWhere = 'iaddpk = ' + TRANSFORM( curSQAcctLoc.iadfk )
    .oDs.Do( 'Query', 'curSQAddress', lcWhere )
  ENDIF
  SELECT curSQAccount
  .SetMode("E")
ENDWITH
```

Notice that in this case we are using the Visual FoxPro `TRANSFORM()` function to convert the key values to their character equivalents for inclusion in the query string. This is fine for numeric data, but for other types we really need to call the *ValToStr()* method to ensure that the data gets formatted correctly for the current data source. The local variable assignment could equally well have been written using that method, as follows:

```
lcCurKey = .oDs.ValToStr( 'curSQAccount', .curkeyval, 'I' )
```

The final call to the form's custom *SetMode()* method ensures that all controls are enabled and that the display is updated.

The *Deactivate()* method must check for pending changes before allowing the user to leave the page and, unless the form is in View mode, it explicitly calls the form's custom *ChkForChanges()* method for each updateable cursor:

```
*** Check for pending changes before the user is allowed to leave this page
*** But not if we are in View Mode
WITH ThisForm
  IF .cMode = 'V'
    RETURN .T.
  ELSE
    llOk = .T.
    llOk = llOk AND NOT .ChkForChanges( 'curSQAccount' )
    llOk = llOk AND NOT .ChkForChanges( 'curSQAcctLoc' )
```

```
    llOk = llOk AND NOT .ChkForChanges( 'curSQAddress' )
    IF ! llOk
      *** We have changes pending
      IF MESSAGEBOX( 'Lose all pending changes?', ;
                     36, 'Uncommitted Changes' ) = 7 && No
        NODEFAULT
        RETURN .T.
      ELSE
        *** Revert Changes
        .Undo()
      ENDIF
    ENDIF
  ENDIF
ENDWITH
```

Note the use of the **NODEFAULT** command to prevent page deactivation when the user does not want to lose pending changes.

Adding new records

The Add button merely calls the form's custom *Add()* method. All this does is lose any pending changes in the cursor and then add a blank record to each cursor. Nothing else needs to be done at this stage because we have nothing to save to our data source. At this stage, it is purely a front-end process.

```
*** Get rid of any changes in cursors and add blank records
*** Account cursor
TABLEREVERT(.T., "curSQAccount" )
APPEND BLANK IN curSQAccount
*** Address Cursor
TABLEREVERT(.T., "curSQAddress" )
APPEND BLANK IN curSQAddress
*** Location Cursor
TABLEREVERT(.T., "curSQAcctLoc" )
APPEND BLANK IN curSQAcctLoc
ThisForm.SetMode( "A" )
```

The final call to the *SetMode()* method sets the form for data entry. (The color convention we use is that fields with yellow backgrounds are mandatory. The code for managing this is defined in the *basectrl::xtxtbase* and the base classes for all other control classes.)

Undoing changes

Like adding a record, this is purely a front-end operation. Uncommitted changes (which are the only ones that we can undo anyway) exist only in the local cursors. The Undo button calls the form's *Undo()* method to revert any changes in the cursors:

```
*** Account cursor
TABLEREVERT(.T., "curSQAccount" )
*** Address Cursor
TABLEREVERT(.T., "curSQAddress" )
*** Location Cursor
TABLEREVERT(.T., "curSQAcctLoc" )
ThisForm.SetMode("E")
```

The call to the *SetMode()* method forces the form into Edit mode and updates the display.

Deleting records

The Delete button calls the form's custom *Delete()* method, which does three things. First, it confirms the user's intention and deletes the current record in each of the three updateable cursors:

```
IF MESSAGEBOX( 'Confirm the deletion of this data', 36, 'Delete Entry' ) = 7
  *** Cancel
  RETURN
ENDIF
WITH ThisForm
  *** Account cursor
  DELETE IN curSQAccount
  *** Address Cursor
  DELETE IN curSQAddress
  *** Location Cursor
  DELETE IN curSQAcctLoc
```

Next, it calls dataset's *TryUpdate()* method to send the changes to the data source. No parameters are actually needed because the method would simply check the dataset's internal collection to build a list of updateable cursors and then try to update them all. However, this does impose an overhead (albeit small), so it is worth passing the names of the cursors explicitly whenever possible:

```
  *** And commit the Delete
  llOK = .oDS.TryUpdate( 'curSQAcctLoc, curSQAddress, curSQAccount'  )
```

Finally, if the deletion succeeds, it re-queries the search cursor and sets focus to the search page. If the deletion fails, it displays a message and reverts the local cursors.

```
  IF NOT llOK
    MESSAGEBOX( 'Unable to Delete', 16, 'Back End Failure')
    *** Display the error log (this is for the developers benefit)
    .oDs.ShowErrors()
    *** Revert the delete
    .Undo()
  ELSE
    *** Re-Query the Search Cursor too
    .oDs.Do( 'ReQuery', 'curAcSearch' )
    *** And return to the first page
    .pgfaccts.ActivePage = 1
  ENDIF
ENDWITH
```

Saving data

This is the most complex piece of code in the form, but only because of the necessity for handling the foreign keys when inserting new records. The Save button calls the forms' custom *Save()* method that first calls the inherited *ChkMandatory()* method to ensure that no fields that have been flagged as required have been left empty. This is a simple piece of pre-

save validation that prevents an unnecessary round trip to the server if a vital piece of data is missing. If this check succeeds a call to the *ValidateForm()* hook method is made. In this case there is no code in that method, and it simply returns True.

```
WITH ThisForm
  IF ! .ChkMandatory()
    MESSAGEBOX( 'One or more mandatory fields (yellow) are empty', ;
                16, 'Cannot Save' )
    RETURN .F.
  ENDIF
  IF ! .ValidateForm( 'save' )
    *** Message handled in ValidateForm()
    RETURN .F.
  ENDIF
```

The next action depends on the form's mode flag. If the form is in Edit mode, then all we need to do is to call the dataset's *TryUpdate()* method and pass the names of the three cursors:

```
  *** OK this far, now try and do the update
  IF .cMode = "E"
    *** Just commit the changes (if any)
    IF ! .oDS.TryUpdate( 'curSQAcctLoc, curSQAddress, curSQAccount'  )
      IF MESSAGEBOX( "Save operation failed. " + CHR(13) ;
            + "Changes on server have been rolled back" + CHR(13) ;
            + "Lose Changes here?", 68, "Save Failed") = 6  && Yep lose them!
        *** Just call the form's Undo() to lose changes
        .Undo()
      ENDIF
    ELSE
      MESSAGEBOX( "Save operation was successful", 64, "Save Succeeded")
    ENDIF
```

In Add mode, things are a little trickier because we need to populate foreign key fields in the *curSQAcctLoc* cursor before we can save it. The first step is to call the *TryUpdate()* method to insert the address record. If that succeeds, we can call the *GetLastID()* function (which is another of the options for the dataset's *Do()* method) to return the PK for the new record.

```
  ELSE
    LOCAL lnAccPK, lnAddPK
    *** We are in Add mode so we will need to get the PKs
    IF ! .oDS.TryUpdate( 'curSQAddress'  )
      IF MESSAGEBOX( "Save operation failed for Address Cursor. " + CHR(13) ;
            + "Changes on server have been rolled back" + CHR(13) ;
            + "Lose Changes here?", 68, "Save Failed") = 6
        .Undo()
      ENDIF
      .LockScreen = .F.
      RETURN
    ELSE
      lnAddPK = This.oDs.Do( 'GetLastID', 'curSQAddress' )
    ENDIF
```

Similar code inserts the record into, and gets the PK for, the Account table:

```
    IF ! .oDS.TryUpdate( 'curSQAccount'  )
      IF MESSAGEBOX( "Save operation failed for Account Cursor. " + CHR(13) ;
            + "Changes on server have been rolled back" + CHR(13) ;
            + "Lose Changes here?", 68, "Save Failed") = 6
        .Undo()
      ENDIF
      .LockScreen = .F.
      RETURN
    ELSE
      lnAccPK = This.oDs.Do( 'GetLastID', 'curSQAccount' )
    ENDIF
```

Finally, we use the retrieved values to update the foreign key fields in the location cursor, and then send it to the back end with a last call to *TryUpdate()*:

```
*** Update Foreign keys in Location Cursor
    REPLACE iacfk WITH lnAccPK, ;
        iadfk WITH lnAddPK ;
        IN curSQAcctLoc
    *** And save it
    IF ! .oDS.TryUpdate( 'curSQAcctLoc'  )
      IF MESSAGEBOX( "Save operation failed for Location Cursor. " + CHR(13) ;
            + "Changes on server have been rolled back" + CHR(13) ;
            + "Lose Changes here?", 68, "Save Failed") = 6
        .Undo()
      ENDIF
    ELSE
      MESSAGEBOX( "Save operation was successful", 64, "Save Succeeded")
    ENDIF
  ENDIF
```

All that is left is to do is to refresh everything, not forgetting to re-query the search cursor in case its result set has been affected by the insert, and put the form back into Edit mode.

```
  *** Re-Query the Search Cursor too
  GO TOP IN curSQAcctLoc
  GO TOP IN curSQAddress
  GO TOP IN curSQAccount
  .oDs.Do( 'ReQuery', 'curAcSearch' )
  *** Force back into Edit Mode
  .SetMode("E")
ENDWITH
```

That's all there is to it!

That really is all there is to it. We hope that you agree that these data classes offer a great deal of power and flexibility, with very little instance-level code required, and that they do a good job of hiding the complexity of working with different databases. One of the biggest benefits that we have found has been that it really is possible to develop an application in Visual FoxPro and, by simply changing the *cCurClass* field in the CURDEFS.DBF table, change the data source that the application uses. There really is no code change required, as you can easily

prove to yourself with the sample form if you have both SQL Server 2000 (or MSDE) and Visual FoxPro available.

Conclusion

In this chapter we have tackled the various ways to access a remote data source and, hopefully, have provided you with a few additional tools and ideas. However, we have not addressed the issue of how to set about building a full-scale client/server application. The reason is simply that the topic really requires a book of its own rather than a single chapter and, fortunately, there is already an excellent one available. If want to know more about building client/server systems with Visual FoxPro, we highly recommend *Client/Server Applications with Visual FoxPro and SQL Server* by Chuck Urwiler, Gary DeWitt, Mike Levy, and Leslie Koorhan, published by Hentzenwerke Publishing.

Chapter 14
VFP and COM

One of the simplest ways in which the functionality and power of Visual FoxPro can be made available to other applications is through the implementation by Visual FoxPro of the Microsoft Component Object Model (COM). Visual FoxPro 7.0 provides the most comprehensive support yet for COM/COM+ and includes many new features that improve the level of integration of components, built using Visual FoxPro, into the COM/COM+ environment. But even as we write, Microsoft has released the new .NET Framework that defines new (and better) standards for component development but which is not directly compatible with COM/COM+. For specific information about using Visual FoxPro with .NET, check out the Web site at www.gotdotnet.com/team/vfp/.

What are COM and COM+?

Before we can start discussing specific tools and techniques, we need to define some of the terminology associated with the world of COM. As Roger Sessions so eloquently stated in his excellent, and highly recommended book *COM+ and the Battle for the Middle Tier* (John Wiley & Sons, ISBN 0-471-31717-9):

> *"… part of the reason it is difficult to pin down COM+ is that Microsoft has frequently abandoned and redefined its terminology. First, Microsoft talked about OLE objects. Then OLE transformed into ActiveX controls. Suddenly ActiveX was as passé as yesterday's top rock band, and Microsoft informed us that all along, it had been talking about COM components. Oops, make that COM components that run in MTS. Did I say MTS? I meant COM+."*

That this trend continues unabated is evidenced by the fact that we have already heard a Microsoft Trainer referring to COM+ as "legacy technology"!

So, COM is...?

The Microsoft Component Object Model (COM) defines a set of standards and mechanisms for creating distributed, binary software components that can interact with each other. Although sometimes referred to as "platform independent", COM is a Microsoft standard and components developed using it have to be compiled as Windows Dynamic Linked Libraries (DLLs). The platform referred to is actually only the development platform, and in reality COM is only "programming language independent." To use COM components directly you must be running a version of Microsoft Windows that supports COM. (This is one of the main reasons why Web Services, which are truly platform independent, are being so widely adopted and supported.)

How does it work?

COM allows an object to expose its functionality to other components and applications by defining both *how* the object exposes itself and *how* this exposure works across processes and

across networks. This is possible because COM specifies that the only way to access, or manipulate, the functionality or data associated with a component is through an *interface* implemented by that component.

Note that the term *interface* in the COM world does not mean quite the same thing that it does when working with Visual FoxPro. For a Visual FoxPro class, its interface is the complete set of PEMs that it defines. For objects the term is usually understood to refer only to an object's "public interface," which is that set of PEMs that it exposes to its environment. On the other hand, a COM interface is a predefined group of related functions that can be thought of as a table of pointers to the individual methods that they represent (this is sometimes referred to as a *Virtual Table* or *vTable*). Furthermore, there is no requirement that a single interface represent the entire functionality of the component, and a single component may (and often does) implement more than one interface. There are several rules governing the definition of COM interfaces, of which the most important are:

- An interface must, ultimately, inherit from the fundamental COM interface *IUnknown.*
- An interface, once defined and published, is immutable. Methods may neither be deleted nor modified in any way.
- Interfaces are descriptive, not definitive. In other words, a COM interface defines method names, their input parameters, return value (if any) and the data types for each, but does not define how methods should be implemented (in other words, no code!).
- A component that implements an existing interface must implement that interface in its entirety.

You can see immediately that if methods in COM interfaces were to actually contain implementation code (as is usual in Visual FoxPro classes), COM would not really be very useful. Instead, COM interfaces can be thought of as a set of "template methods" that are subsequently *implemented* by components. A component *implements* an interface by defining code for each method in the interface and then makes pointers to that code available to the COM library.

Upon creation, each COM component is assigned a globally unique ID (GUID) that follows it around forever and is the unambiguous "name" by which other objects know and address it. This is why components have to be registered before they can be used. Until their GUID has been added to the Registry, other objects have no way of knowing that they exist. It is the COM library (a set of Windows DLLs and EXEs that facilitate the location, creation, and release of COM components) that is accessed by clients that require the services of a component, and it is the COM library that returns the correct pointer when a client calls on a specific method. This (at a very high level) is how COM makes it possible for objects and components in different processes to interact with each other.

And COM+ is...

COM+ is referred to by Microsoft as the "*next step in the evolution of the Microsoft Component Object Model (COM) and Microsoft Transaction Server (MTS)*." It combines

enhancements to the Microsoft Component Object Model with a new version of Microsoft Transaction Server 2.0, together with new services to create a runtime environment for COM components. More importantly, unlike MTS that was an add-on to Windows NT, COM+ is an integral part of Windows 2000-based operating systems.

Basically, COM+ handles many of the resource management tasks that, under COM, had to be handled explicitly by developers (including thread allocation and security) either directly in code or (as in Visual FoxPro) through functionality embedded in their chosen development environment. It automatically makes applications more scalable by providing thread pooling, object pooling, and just-in-time object activation. COM+ also provides transaction support (through MTS) even if a transaction spans multiple databases over a network.

A detailed discussion of COM+ is way beyond the scope of this chapter, but if you are interested in learning more, a great place to start is on the Microsoft COM Technology home page at www.microsoft.com/com. It includes links to white papers, case studies, and detailed information on all of the technologies involved in COM and COM+.

Sounds cool, how could it be "legacy technology"?

As indicated at the start of this section, the evolution of Microsoft's technology and software is a continuing process. The reference to COM+ as legacy technology has to do with the fact that the .NET Framework defines a new set of standards for component development that are different from the existing COM+ standards. A full discussion of .NET is beyond the scope of this book, but suffice it to say that the new standards offer significant improvements over the old by addressing some of the major issues associated with COM+ deployment (for instance, "DLL Hell" and the requirement to explicitly register components). The consequence is that it is possible to deploy a component under .NET by simply copying it to the required host!

While there can be little doubt that, eventually, the new .NET standards will become the de facto standard, for the foreseeable future it is certain that Microsoft will continue to support COM+ alongside .NET component standards (if only because much of Microsoft's own software is founded on COM+, and it will take Microsoft a significant amount of time to convert everything to the new standard). To overcome the differences, the .NET Framework includes a Type Library Importer (TBLIMP.EXE) that converts a COM+ type library into an equivalent .NET "manifest" by creating a proxy called a *Runtime Callable Wrapper* (RCW). It is this RCW that allows a COM+ component to be accessed from .NET code.

All about interfaces

The key to understanding how COM works is understanding the role of COM interfaces. As we hinted previously, COM interfaces are not the same thing as we usually understand in the context of Visual FoxPro classes. The problem with interfaces is that in order to access a component's interface, a client application has to know how to address it (the methods to call, the parameters to pass, and the return values to expect).

Late binding

One solution is to have some way of discovering, at run time, what interfaces an object supports and how to call them. This is what *Automation* is all about (which, in the spirit of change that appears to pervade this subject, was not so long ago known as *OLE Automation*). The technology was originally developed as a result of the requirement to create generic macro

and scripting languages that could be used to control applications. Clearly such languages could not incorporate every possible interface for every possible component, so some other solution had to be found.

The solution was to define a single standard COM interface, named *IDispatch*, that can be used to access an object. By implementing *IDispatch*, a component can expose any number of properties, methods, and events to its clients through a single method of the *IDispatch* interface, named *Invoke*. Conversely, clients can discover whether a component supports a given piece of functionality (and how to call it) by calling another method in the *IDispatch* interface, named *GetIdsOfNames*. Components that support the *IDispatch* interface are, therefore, said to be "automation aware."

This has one important consequence for Visual FoxPro developers. Both the CreateObject() *and* GetObject() *functions require that the target object implement* IDispatch*. If it doesn't, you get a "No such interface supported" error. In other words, Visual FoxPro can only instantiate automation aware components using these functions. To create an instance of objects that do not support* IDispatch*, you must use* CreateObjectEx() instead*. (See the section "An overview of the SOAP toolkit" in Chapter 5 for an example.)*

The process is that whenever an automation aware object is instantiated, it returns a handle to its *IDispatch* interface. The client uses this to call the *GetIdsOfNames()* method, passing the name of the required property or method. The component then checks itself to see whether the required functionality is supported and, if so, returns a numeric *Dispatch ID* (*DispID*) that identifies the specific internal method to call. Next, the client uses this *DispID* in a call to the *Invoke()* method of *IDispatch* passing any necessary parameters in a standard structure. The component uses the *DispID* from the client (that it sent to the client in the first place, remember) to identify which internal function it is going to call. The parameter structure from the client is disassembled, checked, and re-assembled as the appropriate internal instruction. Any results are repackaged and returned to the client as the result of the call to *Invoke()*.

However, as always in programming, there is no such thing as a free lunch. Late binding involves a considerable overhead. As you will have realized already, each method call actually requires *two* calls to the component, first to get the *DispID* and then to actually call the *Invoke()* method. Since each call must cross the boundary between the client application and the COM component, it can be (relatively) slow.

Fortunately, in Visual FoxPro we do not need to worry about any of this; it is all handled automatically by the compiler and interpreter when we define and create objects using the *CreateObject()* function. For example, if we want to create, and use in code, a late bound reference to Microsoft Word we can write code like this:

```
loWord = CREATEOBJECT( 'word.application' )
WITH loWord
  loWord.Documents.Open( "some.doc" )
  ...
ENDWITH
```

Early binding

The other solution to the problem of resolving interface references is to simply hard-code the *DispID* into the client application. If this can be done, the executable code only needs the code to call the object's *Invoke* method, thereby removing one complete set of calls. This is called "early binding" and is by far the most efficient way to do things. The impact of early binding on performance can be dramatic when dealing with setting or retrieving a number of properties, or calling short methods, because in such circumstances the overhead of retrieving the *DispID* for each call can be a significant part of the total execution time.

However, since early binding involves hard-coding the *DispID* for each call at compile time, code that relies on it will break if the *DispID*s for the server changes (for example, when a different version gets installed, or the component IDs get re-generated). For this reason, early binding is really only appropriate when dealing with servers that are likely to remain stable.

Version 7.0 of Visual FoxPro introduced an additional parameter to the *CreateObjectEx()* function so that it can be used to generate an early bound reference. The new, third, parameter specifies the ID of the interface to which a reference is required. However, if this parameter is passed as an empty string, a reference to the object's default interface (IID) is returned. So to create, and use in code, an early bound reference to Microsoft Word we can simply write code like this:

```
loWord = CREATEOBJECTEX( 'word.application', "", "" )
WITH loWord
  .Documents.Open( "some.doc" )
  ...
ENDWITH
```

VFP 7.0 also introduced the *GetInterface()* function that returns an early bound reference to an interface in a COM object. This can be used to implement early binding for objects to which a reference is only obtainable at run time. It allows you to retrieve either a reference to the default interface for the object, or to specify a specific interface name and, optionally, the type library. For full details of this function see the Visual FoxPro on-line documentation.

In order for early binding to work, we require some way to determine the *DispIDs* of the methods exposed by an object. It would also be useful if, at the same time, it was possible to validate that the individual method's parameter requirements were being met (to prevent calling errors at run time). What is needed, therefore, is a complete description of the methods exposed by the component and their *DispIDs*. This description is stored in a special file named a "Type Library."

How does this apply to Visual FoxPro?

Components created in Visual FoxPro support both early (*vTable*) binding and existing late (*IDispatch*) binding and so are said to exhibit "dual-interface support". However, it is important to realize that while Visual FoxPro servers support both interfaces, the one that is actually used at run time depends entirely upon the client. As noted earlier, the ability of a client to use early binding depends upon it having access to a type library. So, the obvious question is, how do we define interfaces and then create Type Libraries for our components?

The component support files

Whenever you build a COM component (either as an EXE or DLL), two additional files are created automatically. The first, normally assigned a TLB extension, is the Type Library; the second, normally assigned a VBR extension, contains the Registry information generated for the component. Note that although the Type Library is created as a freestanding file by default, it may also be bound directly to the EXE or DLL. (This is why there are apparently no Type Libraries for many COM servers.)

Interfaces and type libraries (*Example: EGComif.pjx, EGComif.prg*)

COM Interfaces are actually defined using an *Interface Definition Language* (IDL), which is a C++ like language that defines the syntax, parameters, return values, and Help context IDs for an interface. (For details of the Microsoft Interface Definition Language (MIDL) see http://msdn.microsoft.com/library/en-us/midl/midlstart_4ox1.asp.) The Type Library is a language independent binary representation of the actual IDL code.

Of course, a Type Library is not directly readable by us humans (unless you happen to be able to read binary code), so in order to view it we need to use a tool. The obvious choice in Visual FoxPro 7.0 is the Object Browser that was added to the product for precisely this purpose. However, both Visual Studio and Visual Studio .NET include a COM server that can read type libraries (TLBINF32.DLL). If you really must reinvent wheels, or if you do not have Visual FoxPro 7.0 or later, you can use this component to create your own "Type Library Browser."

Fortunately for us, as Visual FoxPro developers, we do not need to worry about using an IDL or creating Type Libraries. We can simply define interfaces using standard Visual FoxPro syntax and allow Visual FoxPro to worry about creating the necessary Type Library for us. The following code illustrates the process. First we define a class as **OLEPUBLIC** with the exposed methods and properties that we want in its interface:

```
************************************************************************
* Program....: egcomif.prg
* Compiler...: Visual FoxPro 07.00.0000.9465
* Purpose....: Simple COM class Interface definition
************************************************************************
DEFINE CLASS egComIF AS session OLEPUBLIC
  *** Add an exposed property [WILL appear in the Type Library]
  cExpProp = ""
  *** And a protected property [Will NOT appear in Type Library]
  PROTECTED nHidProp
  nHidProp = 0

  ****************************************************************
  *** [E] EXACTSEEK(): Runs a SEEK inside an EXACT setting
  *** [This method WILL appear in the Type Library]
  ****************************************************************
  FUNCTION ExactSeek( tuValue AS Variant, ;
                      tcAlias AS String, ;
                      tcTag AS String ) AS Variant ;
                      HELPSTRING "Runs a SEEK inside an EXACT setting"
  ENDFUNC
```

```
  ************************************************************************
  *** [P] SETUP(): Set up working environment
  *** [This method will NOT appear in the Type Library]
  ************************************************************************
  PROTECTED FUNCTION SetUp()
    *** Need to set Multilocks if we want buffering!
    SET MULTILOCKS ON
  ENDFUNC

  ************************************************************************
  *** [P] INIT(): Standard Initialization method
  *** [Native PEMs for Session Class do NOT appear in Type Library]
  ************************************************************************
  FUNCTION Init
    RETURN This.SetUp()
  ENDFUNC

ENDDEFINE
```

Note that we are using the *Session* base class for our server. This is no accident. This base class was modified in Visual FoxPro 7.0 specifically to improve its usefulness as the root class for creating COM servers. First, its native PEMs are no longer written out to the Type Library, so only custom PEMs that are defined as "Public" show up. Second, when a private datasession is specified, **EXCLUSIVE**, **TALK**, and **SAFETY** are all defaulted to **OFF**. (Note: A strange omission, in Visual FoxPro 7.0, is that **SET MULTILOCKS** is still defaulted to **OFF**, which means that you must explicitly set it **ON** before you can use any form of buffering.)

This definition is stored in a PRG file, which is the only member of the *egcomif* project. We simply built the project as a "Multi-threaded COM server (dll)". On completion of the build there are three new files in the directory:

- EGCOMIF.DLL The COM server itself
- EGCOMIF.TLB The Type Library
- EGCOMIF.VBR The registration information file (see the next section)

If we now open the type library for our new DLL in the Object Browser (see **Figure 1**), we can examine its contents in detail. Notice that the class name has been prefixed with the letter "I" and used as the name for the interface in this server (*Iegcomif*). Notice also that *Iegcomif* automatically inherits the two basic COM interfaces (*IDispatch* and its parent *IUnknown*).

The single exposed method that we defined (*ExactSeek()*) appears in the list of methods for the *Iegcomif* interface, while its calling prototype appears together with the Help text that we specified in the bottom pane. Neither the native Session class methods, nor the protected method that we defined (named *SetUp()*) appear. However there *are* seven other methods that we did not define and that do not look like normal Visual FoxPro methods. These are the methods that our component inherits from *IUnknown* and *IDispatch*. **Table 1** lists their names and functions, and this really is all that we need to know about them because we never need to work with them directly.

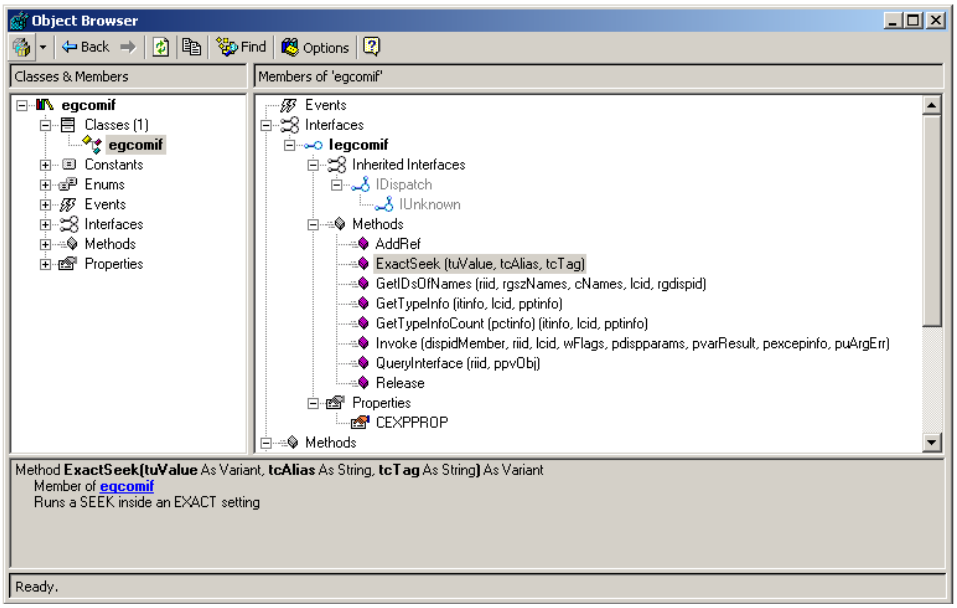


***Figure 1**. Simple COM server interface.*

***Table 1**. The methods of the IUnknown and IDispatch interfaces*

Method	Description
IUnknown	
QueryInterface	Returns pointers to supported interfaces
AddRef	Increments reference count
Release	Decrements reference count
IDispatch	
GetTypeInfoCount	Retrieves the number of type information interfaces that an object provides (either 0 or 1)
GetTypeInfo	Gets the type information for an object
GetIDsOfNames	Maps a single member and an optional set of argument names to a corresponding set of integer DISPIDs
Invoke	Provides access to properties and methods exposed by an object

The exposed property (*cExpProp*) appears in the properties section of the interface, but, as with the methods, neither the native Session class properties, nor our protected property (*nHidProp*) are visible. Since neither *IUknown* nor *IDispatch* defines any additional properties, all that our type library shows is the one exposed property that we defined.

In Visual FoxPro, on machines running Windows NT 4.0 or later, Type Libraries are automatically included in the DLL or EXE file as a bound resource when the component is built. This eliminates the need to ship the extra TLB file, although it is still built, along with the VBR file, which is only needed when deploying components remotely.

Registration information file

The other file created when we built our project was the registration information file (EGCOMIF.VBR). A VBR file lists the globally unique IDs (GUIDs) that have been assigned to the classes and interfaces defined in your component. It is structurally similar to a REG (Windows Registry) file but without the hard-coded paths and defines the following:

- Header information (that is, language and version)
- Keys for the component (HKEY_CLASSES_ROOT entries)
- Keys for each class within the component (HKEY_CLASSES_ROOT\CLSID entries)
- Keys for each interface of each class (HKEY_CLASSES_ROOT\INTERFACE entries)

Figure 2 shows the VBR file generated for our simple example.

```
VB5SERVERINFO
VERSION=1.0.0

HKEY_CLASSES_ROOT\egcomif.egComIF = egcomif.egComIF
HKEY_CLASSES_ROOT\egcomif.egComIF\NotInsertable
HKEY_CLASSES_ROOT\egcomif.egComIF\CLSID = {EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02}
HKEY_CLASSES_ROOT\CLSID\{EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02} = egcomif.egComIF
HKEY_CLASSES_ROOT\CLSID\{EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02}\ProgId = egcomif.egComIF
HKEY_CLASSES_ROOT\CLSID\{EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02}\VersionIndependentProgId = egcomif.egComIF
HKEY_CLASSES_ROOT\CLSID\{EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02}\InProcServer32 = egcomif.dll
HKEY_CLASSES_ROOT\CLSID\{EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02}\InProcServer32\"ThreadingModel" = Apartment
HKEY_CLASSES_ROOT\CLSID\{EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02}\TypeLib = {00907196-B098-4E55-996B-D1F76C6329BD}
HKEY_CLASSES_ROOT\CLSID\{EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02}\Version = 1.0
HKEY_CLASSES_ROOT\CLSID\{EC7A8104-DEB0-4C1E-9E6A-F52AA128FA02}\Foxruntime = VFP7T.DLL
HKEY_CLASSES_ROOT\INTERFACE\{45CC688A-989F-43C6-9968-CADA3DB0BEF9} = egComIF
HKEY_CLASSES_ROOT\INTERFACE\{45CC688A-989F-43C6-9968-CADA3DB0BEF9}\ProxyStubClsid =
{00020424-0000-0000-C000-000000000046}
HKEY_CLASSES_ROOT\INTERFACE\{45CC688A-989F-43C6-9968-CADA3DB0BEF9}\ProxyStubClsid32 =
{00020424-0000-0000-C000-000000000046}
HKEY_CLASSES_ROOT\INTERFACE\{45CC688A-989F-43C6-9968-CADA3DB0BEF9}\TypeLib = {00907196-B098-4E55-996B-D1F76C6329BD}
HKEY_CLASSES_ROOT\INTERFACE\{45CC688A-989F-43C6-9968-CADA3DB0BEF9}\TypeLib\"Version" = 1.0


; TypeLibrary registration
HKEY_CLASSES_ROOT\TypeLib\{00907196-B098-4E55-996B-D1F76C6329BD}
HKEY_CLASSES_ROOT\TypeLib\{00907196-B098-4E55-996B-D1F76C6329BD}\1.0 = egcomif Type Library
HKEY_CLASSES_ROOT\TypeLib\{00907196-B098-4E55-996B-D1F76C6329BD}\1.0\0\win32 = egcomif.dll
HKEY_CLASSES_ROOT\TypeLib\{00907196-B098-4E55-996B-D1F76C6329BD}\1.0\FLAGS = 0
```

***Figure 2**. VBR file for the egcomif class.*

The last part of the registration file lists the registration information for the associated Type Library. The information that is written to the VBR file is also used to automatically register the component on the host machine when the EXE or DLL is built. This makes it easier to test components, but also means that you should not check the "Regenerate Component IDs" option to avoid creating multiple Registry entries for the same component.

In other words, you should only regenerate the component IDs when you want to create a new version of the component (that is, one with a different set of GUIDs) that can be installed alongside existing version(s) rather than simply replacing them. Note that this "side-by-side" approach, which was intended to preserve backward compatibility while still allowing progress, is actually the main cause of "DLL Hell." As noted elsewhere, the whole mechanism has been re-designed for the .NET Framework, which uses a different way of handling component registration.

Working with COM in Visual FoxPro

Visual FoxPro is an ideal tool for creating COM components because it hides most of the complexity of creating and registering components. In fact, the only time that there is any difference between a class intended for exclusive use in Visual FoxPro and one intended to create a COM component is when it is actually built into an EXE or DLL. The inclusion of the key word `OLEPUBLIC` in a class definition is the indicator to the compiler that the class has to be built as a COM component and, as we have seen, it handles the generation of the type library and registration files automatically.

Visual FoxPro can be used to create two types of component, as follows:

- **Automation servers**—An Automation server is an application that runs in its own process space, which is why it can also be referred to as an "out of process server." It can provide services, including user interface elements, to other applications. To create an Automation server, build a project as "Win32 executable / COM server (EXE)."
- **COM servers**—A COM server is a DLL that runs in the client's process space, which is why it can also be referred to as an "in-process server." It defines an object that exposes methods, but cannot have any user interface elements. Visual FoxPro supports both single-threaded DLLs (that use the VFP7R.DLL run-time library) and multi-threaded DLLs (that use the VFP7T.DLL run-time library).

Visual FoxPro 7.0 includes several well-documented samples that illustrate the construction and use of Automation servers. Such applications are very specialized and we will say no more about them here. For details see the "Server Samples" topic in the Visual FoxPro 7.0 Help files. For the remainder of this chapter we will concentrate on COM servers, which are deployed as DLLs.

What's the difference between single and multi-threaded DLLs?

As implied by the name, single threaded components execute all code in a single processing "thread." Basically this means that each instance of the object can only ever be executing one method at any given time. The COM runtime environment deals with this situation automatically by *serializing* requests. In other words, when an object is executing code, any calls to it are queued. As the object completes processing one call, the next is executed, and so on until the queue is empty.

There are times when it is imperative that components complete one task before being allowed to proceed with the next, and that scenario is one where you might consider using a single-threaded component. However, the big problem with single-threaded components is that they are prone to "request blocking" when methods on the component have significantly different execution times.

Consider what happens when two users are accessing the same instance of a single-threaded component. User A calls the *ProcessAll* method (that is known to take several minutes to run) and happily goes off to get a cup of coffee. Meanwhile, User B needs to call the *LookUp* method, which takes only a few milliseconds to execute. Unfortunately, poor User B has to wait several minutes until User A's processing is complete before his call can even be

submitted to the component. Single-threaded components are, therefore, said to scale poorly because they cannot easily support additional users.

Multi-threaded components, as the name implies, can have more than one processing thread and so can execute more than one method at a time by simply creating new threads as needed. The result is that they are much less prone to request blocking, although, because of the necessity to ensure that individual threads do not interfere with each other, they do run a little more slowly than their single-threaded equivalents. In practice the benefits of scalability, and the ability to access COM+ services, mandate the use of the multi-threaded model in all but the most specialized of situations.

Why are there two versions of the Visual FoxPro runtime library?

The answer is to support the two different threading models. The single-threaded runtime, VFP7R.DLL, which was all that was available prior to Version 6.0 Service Pack 3, provides services for the full range of application types that can be created using Visual FoxPro:

- Win32 executable (EXE)
- Visual FoxPro application (APP)
- Out-of-process servers (EXE)
- In-process servers (DLL)

However, in the context of COM, this runtime is limited because it cannot service multiple in-process servers (see "A brief overview of threading" later in this chapter for details of why this is so), and so each COM server must have its own separate instance of the library. This is managed by creating temporary copies of the entire runtime library on disk as follows:

- The first component to be instantiated is assigned exclusive use of the default VFP7R.DLL runtime library.
- As other components are instantiated, the VFP7R.DLL file is copied and assigned a new name derived from the name of the component's DLL. Note that this also happens when a Visual FoxPro EXE instantiates an object defined as a Visual FoxPro DLL because both the EXE and the object require the same runtime library.

As noted earlier, single-threaded DLLs scale poorly because of request blocking and cannot be used in COM+ environments (which require multi-threaded support). When creating an in-process server, the preferred option is multi-threaded, which requires the use of VFP7T.DLL. However, while this library helps to eliminate the request blocking issues, and implements Apartment Threading, it is intended only for use with in-process servers. Consequently, some functionality that requires either direct interaction or visual representation has been disabled (the Table and Report Designer, for example). Note that you can still use visual classes in components created as multi-threaded DLLs, you simply cannot make them visible.

The result is that the VFP7T.DLL is smaller (by about 350KB) than its single-threaded cousin. The most important issue, however, is that it does not need to have individual copies of

the Visual FoxPro runtime library and can, therefore, participate fully in the COM+ environment.

How does COM work?

Unless you have a particular interest in this subject, this section can be safely skipped (the real story continues with the section entitled "What is 'instancing'") because Visual FoxPro handles all of the issues transparently for us when creating COM components. However, the whole subject of threading is surrounded in confusion and mystery, and it seems that no two sources agree on the precise meaning of the terminology, which is, in any case, pretty confusing to begin with. For instance, a "free-threaded" server still uses the "apartment threading model," and the "multi-threaded VFP runtime" is neither more, nor less, multi-threaded than the single-threaded one.

There are several good books on this subject, but if you are really interested in the low-level details, we suggest that *Designing Component Based Applications* by Mary Kirtland, published by Microsoft Press, is a good (and very readable) place to start.

Of processes, threads, and variables…

Let's begin by getting a working definition for some terms (these may not be rigorous, but they will suffice for the purposes of this discussion):

- **Process**—A Windows process consists of an address space in which an application or program is actually run. It provides access to memory and system functions including screen handling and disk I/O. Processes are strictly isolated. In other words, one process may not directly access another, and as a consequence all Inter-Process Communication (IPC) must be handled by Windows itself. If you like, you can visualize a process as a "virtual PC."
- **Thread**—A thread is an execution path inside a process. It is where the code is actually processed, and each process must have at least one thread. However, all threads in a single process must share the same set of resources (for instance, memory) so, although multiple threads may exist, they are all competing for and accessing the same set of resources. If the process is a virtual PC, a thread is a "virtual CPU."
- **Stack variable—**The stack is an area of memory that is reserved for use by the currently executing function. Values are added to and retrieved from the stack in Last-In-First-Out order. When a function completes, its stack is released. In Visual FoxPro terms, stack variables are "Local."
- **TLS variable—**Thread Local Storage (TLS) is a special area of memory reserved for use by an individual thread and accessible only from within that thread. However, its use requires a special set of API calls and there is a performance penalty associated with it. In Visual FoxPro terms, TLS variables are "Private."
- **Heap variable—**The heap is an area of memory that is reserved for use by the current process. Variables stored in the heap persist for the life of the process and are available to all threads in the current process. In Visual FoxPro terms, Heap variables are "Public (Global)."

You can now see why, and how, single and multi-threaded components differ. By limiting processes to a single thread, all of the issues of internal competition for resources, and the necessity to deal with variable scoping, are removed. Only one thread at a time can exist, so there is never any possibility of conflict. Of course, such a limited use of the process's resources is both inefficient and prone to blocking. It is, in Visual FoxPro terms, like creating a dedicated single-user application.

Conversely, if we are to allow multiple threads in a process, we must ensure that variables are correctly scoped to avoid conflicts between threads, or having a value created by one thread overwritten by another. There is, in fact, a very close parallel between the issues surrounding multi-threading and those surrounding multi-user access to a database. In Visual FoxPro, such issues are largely handled automatically. For example, when you issue a **REPLACE** statement, Visual FoxPro places and releases the necessary locks, and when you declare a variable as **LOCAL**, Visual FoxPro handles its memory allocation and access correctly.

In the context of threads, the operating system provides a variety of tools and functions to address the problems, but there is no automatic handling like Visual FoxPro provides. Code that properly handles the issues of multiple thread execution is referred to as "thread safe", but creating thread safe code directly in a language like C++ involves an awful lot of work and is extremely tricky. One of the benefits of using the COM framework is that it makes handling the task of creating thread safe components much easier. The major benefit of using Visual FoxPro to create multi-threaded components is that we can leave it to Visual FoxPro to worry about all this stuff!

How EXEs and DLLs differ

The basic difference between an EXE and a DLL is that executing an EXE always creates a new process. The EXE file is loaded into the memory area allocated to the new process and a new thread is created for the designated main program, which is immediately started. (This is why you always have to have something that is capable of being run, either a form or a PRG, identified as the "main" item in a Visual FoxPro project.) Next, any required libraries, localization, and configuration files have to be located and loaded.

This all makes loading up an EXE for the first time a comparatively slow process. You can see this when you instantiate an application like Word, or Excel, for Automation using the **CREATEOBJECT()** function. The first time you do it there is a noticeable delay, but if you release the object, and re-instantiate it, the second time is much faster because the necessary associated files are already loaded.

Unlike an EXE, a DLL can only be loaded into an existing process and cannot be launched directly. As a result, there is little overhead associated with loading it. Moreover, Windows does not allow the same DLL to be loaded more than once into any given process. Should a process attempt to re-load an existing DLL, Windows simply increments its internal usage counter and simply returns a reference to the existing instance. Similarly, releasing a DLL merely decrements the instance counter—the DLL is only released from memory when the instance counter reaches zero. This all means that DLLs are, generally, faster to load than EXEs. There is, however, a catch (isn't there always?).

The problem with DLL caching

This form of instance management dates back to the days before Windows 95 when processes were effectively single-threaded, and it has a major implication for data handling in DLLs that

must run in a multi-threaded environment. This is because, as indicated earlier, in order for a variable to be declared as global in the DLL, it has to be handled as a Heap variable and is, therefore, also global to the entire process in which the DLL is running. As long as the process has only one thread, or the DLL in question is merely a static procedure library that manages its own data internally as a "black box," this isn't really a problem.

However, the introduction of multiple threading made it entirely possible that a single process could contain two (or even more) components, each of which relied on the *same* DLL. The effect of multiple DLLs addressing, and changing, the same global variables had a dramatic, and often unforeseen, effect on applications (usually resulting in a crash!). So, if only for this reason, the issue of thread safety has to be taken seriously in the context of designing and implementing a multi-threaded DLL. Fortunately for us, Visual FoxPro 7.0 takes care of this for us.

Of threads and apartments

You may be thinking, about now, that there is a small loophole in the COM model (don't worry if you hadn't realized it yet). What happens if two objects, each running in different threads, attempt to call the same single-threaded object simultaneously? Since it is single-threaded, it can only execute one method at a time, but it is now being asked to handle two calls simultaneously. As you can see, the most likely result is a sudden attack of schizophrenia, resulting in an error.

Fortunately, COM provides an escape mechanism—the "Apartment." An apartment consists of one or more threads and an invisible window (honestly, it does!) whose purpose is to act as a message queue. Apartments with only one thread are imaginatively called "Single-Threaded Apartments" (STA) and those with more than one thread are "Multi-Threaded Apartments" (MTA).

STAs explicitly bind a message queue and a specific thread for their lifespan, and components based on STAs are, therefore, referred to as "Apartment Threaded." A single process can have as many STAs as it can support threads, and the first STA to be created is known as the "main STA."

Conversely, MTAs are not bound to a specific thread and, instead, allocate threads dynamically to methods on an as-needed basis. This is referred to as "Free Threading" and allows MTAs to deal with multiple requests concurrently. However, a single process can only have a single MTA. The threading model required by a component is defined when it is created and is part of its registration information (see the *ThreadingModel* key in Figure 2). This allows Windows to load COM objects into apartments of the required type.

Each apartment (whether STA or MTA) actually defines a calling boundary. More than one object can share an apartment, and objects in the same apartment can call each other's methods directly. However, calls between objects in different apartments must be handled through Windows itself.

This simple design ensures that STAs are thread-safe because all objects sharing the apartment can use only that one thread. The result is that, irrespective of the number of objects, only one method call will ever be executing at any time. Conflicts are simply not possible. When an object in another thread needs to access a method of an object in a different apartment, Windows intercepts the call, packages it up, and sends it to the apartment's message queue where it waits until the thread is idle. Waiting messages are unpacked and translated into the appropriate internal call. This whole process for synchronizing calls is

referred to as "Marshalling" and, since message queues work on a "First-In-First-Out" (FIFO) basis, it ensures that calls are always processed in the order in which they were received.

Note that it is the host application that is responsible for providing STAs in which objects are created and that an in-process STA component will use whatever STA its host application provides. Visual FoxPro applications do not create a new STA for each object, which explains why, even when using COM servers in a Visual FoxPro application, you do not gain scalability—all components are running in the same apartment anyway. Specialized host applications like Internet Information Services (IIS) or Microsoft Transaction Server (MTS) do create STAs as necessary and objects are automatically loaded into their own STA, thereby providing the necessary scalability.

The evolution of COM in Visual FoxPro

The first version of Visual FoxPro that allowed developers to actually create COM servers directly was Visual FoxPro 5.0 but, while Visual FoxPro 5.0 was multi-threaded, its runtime was not thread safe! As a consequence, only one instance of the runtime could be permitted in any process and so it had to be loaded into the main STA. No other instances were permitted, and attempting to load a second instance of the runtime caused an error.

Since the runtime was loaded into an STA, and only one instance was permitted to a process, the consequence was that only one method of one COM object could ever be executing at any time and all other calls were blocked. This limitation severely restricted the usefulness of COM DLLs built using Visual FoxPro 5.0.

Apartment threading was introduced into Visual FoxPro with Version 6.0, but although COM objects were loaded into separate apartments, the Visual FoxPro runtime DLL itself was not and it was no more thread-safe than its Version 5.0 predecessor. In order to avoid the DLL caching issues described earlier, Visual FoxPro 6.0 uses a simple trick to fool Windows.

When a Visual FoxPro DLL is loaded, it first searches for other Visual FoxPro components in the same process. If one is found, the DLL creates a copy of itself and adds an auto-incrementing suffix to its name (XXXR1.DLL, XXXR2.DLL, and so on). When the DLL is released, this temporary file is deleted but, because Windows relies on the name of the DLL for its caching, this trick ensures that each component gets its own instance of the runtime library.

This solved the problems associated with Visual FoxPro 5.0 and made it possible to load more than one DLL into a process. However, it was not the final solution because as mentioned previously, it is the object that gets loaded into an apartment, not the DLL. In order to avoid issues when multiple components exist in different threads, the entire runtime library is locked whenever a COM component is executing a method. So while it is possible to have multiple components in different DLLs executing in different threads at the same time, only one component from each DLL can ever be executing code at any time. This limited the scalability of Visual FoxPro 6.0 components.

The current state is that Visual FoxPro 7.0 supports (and extends) the "multi-threaded DLL" that was introduced with Service Pack 3 for Visual FoxPro 6.0. In fact, this DLL is no more multi-threaded than any previous one, but its runtime library is now thread-safe. The multi-threaded runtime implements Thread Local Storage (TLS) so that the same DLL can be executed in different threads. Finally, Visual FoxPro allows us to create objects in different apartments from the same DLL that remain independent of each other.

Note that although the VFP7T.DLL provides thread safety, it does not provide data safety! Objects in the same thread, instantiated from the same DLL, all share the same Visual FoxPro datasession. Therefore they have the potential to trample on each other's data. The solution is to ensure that objects inherit from a class that allows each to have a Private Datasession. It is, therefore, no coincidence that the Session base class was introduced at the same time as support for multi-threaded DLLs. The Session class delivers a private datasession without incurring the overhead of using a Form or Toolbar class and for this reason is the preferred base class for creating COM components.

What is "instancing"?

The Servers tab of the Project Info dialog (see **Figure 3**) includes a dropdown list where the instancing mechanism can be specified for each class. "Instancing" describes the rules that govern when and how the server is instantiated. Visual FoxPro supports three instancing modes:

- Not Creatable (Mode = 0)—Specifies that this server can only be created inside Visual FoxPro (that is, not available for Automation).
- Single Use (Mode = 1)—This defines a server that may be created inside Visual FoxPro and also as an Automation server. Each request for use of a single use server requires that a fresh copy of the server be started.
- Multi Use (Mode = 2)—The default setting, this also defines a server that may be created both inside Visual FoxPro and as an Automation server. However, instead of creating a fresh copy of the server for each new request, Automation clients receive instead a reference to the existing instance.

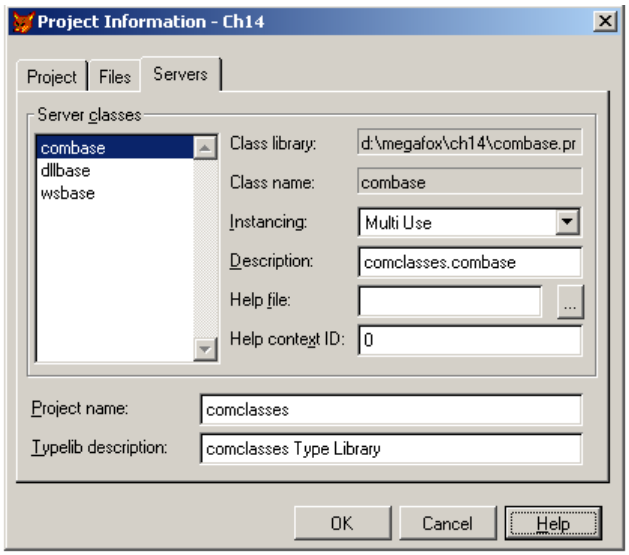


***Figure 3**. Setting component instancing.*

Note that the Single Use setting is really only applicable in the context of Automation servers (EXE). When set to Single Use, each request for an Automation server initiates a new Windows process. With Multi Use servers, only the first request generates a new process. All subsequent requests are handled by the existing process.

Conversely, multi-threaded DLLs simply ignore the setting of the instancing property and are always Multi Use. For single-threaded DLLs, setting the Single Use property means that any attempt to instantiate more than one object from the DLL will result in an error. Note also that Microsoft Transaction Server always requires Multi Use instancing, so Single Use DLLs cannot be used with COM+.

In practice, therefore, Multi Use instancing is the norm for DLLs although Single Use instancing does have a specific role in the context of running high-risk operations. By ensuring that such operations are always running in separate processes it effectively isolates them. If a Single Use instance crashes, it is the only one affected, and other processes can continue.

How do I create a COM DLL?

The actual process of creating a COM component in Visual FoxPro is very simple indeed. The first thing that is needed is to create a project that contains at least one class that is defined as **OLEPUBLIC**. This can be done programmatically by including the keyword in the **DEFINE CLASS** statement as follows:

```
DEFINE CLASS SimpleDLL AS SESSION OLEPUBLIC


  ************************************************************************
  *** INIT(): Standard Initialization method
  ************************************************************************
  FUNCTION Init
    RETURN This.SetUp()
  ENDFUNC

  ************************************************************************
  *** [P] SETUP(): Set up working environment
  ************************************************************************
  PROTECTED FUNCTION SetUp()
    *** Need to set Multilocks if we want buffering!
    SET MULTILOCKS ON
  ENDFUNC

  ************************************************************************
  *** CALCULATE(): Carries out the specified calculation
  ************************************************************************
  FUNCTION Calculate( tnVar1 AS Number, ;
                      tnVar2 AS Number, ;
                      tcOp AS Character ) AS Number ;
           HELPSTRING "Returns result of specified operation on input values"
    lnResult =  EVALUATE( TRANSFORM(tnVar1) + (tcOp) + TRANSFORM(tnVar2) )
    RETURN lnResult
ENDDEFINE
```

If using the visual class designer to create components, a different base class must be chosen (Session is a non-visual class), and the Class Info dialog has a checkbox that is used to define the class as **OLEPUBLIC** (see **Figure 4**).

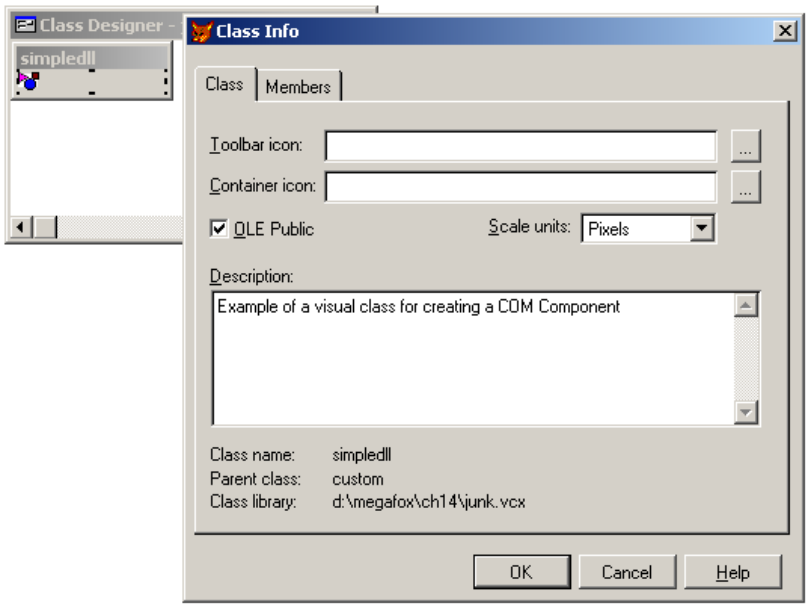


***Figure 4**. Defining a COM class in the visual class designer.*

Unless there is some particular reason to do otherwise, we strongly recommend using the Session class as the basis for all your COM DLLs. The reasons are that the Session class has features specifically designed for use as the basis for COM components:

- **Native PEMS do not appear in the Type Library**. If you use any other class, all the native Visual FoxPro properties, events, and methods will be exposed in the resulting component's interface and type library. You need to manually set the status of every property, event, and method that you do not want exposed in the component's interface.
- **Provides a private datasession**. In order to ensure that data integrity is preserved, you should ensure that your components always create their own private datasession. You could, therefore, use a Form or a Toolbar as the root class for your components, but these have more overhead associated with them and since a DLL cannot have a visual component, there is little point in using a class that is designed for visual display.
- **Default settings more appropriate**. The default settings for several options that are scoped to datasession have been altered in the Session class so that you do not need to worry about explicitly setting them. Specifically, `EXCLUSIVE`, `TALK`, and `SAFETY` are all defaulted to `OFF`. However, a strange omission, in Visual FoxPro 7.0, is that `SET MULTILOCKS` is still defaulted to `OFF`, which means that you must explicitly set it `ON` before you can use any form of buffering in your components.

Ensure that the class definition is the "main" program in the project (typically the project has only one class, so it will be set as main by default when you add it). Set the instancing requirement (the default will be Multi Use) and any other optional information (descriptions, associated Help file, Help context ID) in the Project Info dialog (Figure 3) and then build the project just like any other Visual FoxPro project using the appropriate threading option from the Build dialog (see **Figure 5**).

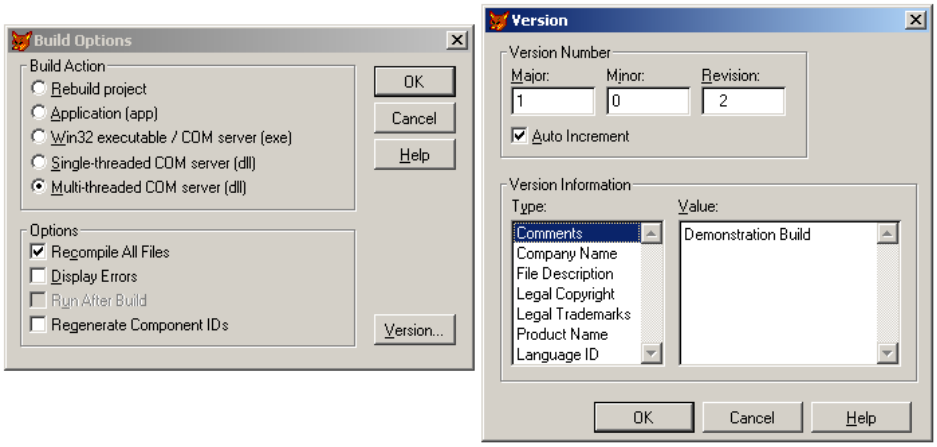


***Figure 5**. Project build options and version information.*

The build process creates and automatically registers the DLL on the machine used to build it. It also creates the Type Library and Registration file (see "The component support files" section earlier in this chapter for details).

All that is left to do is to test your new DLL. To create an instance of the DLL, use the standard Visual FoxPro `CREATEOBJECT()` function using the file name that you defined for the DLL and the name of the `OLEPUBLIC` class within the DLL as the input parameter. Thus, if we had created the *SimpleDLL* class defined earlier in this section as part of a DLL named "TestCOM", the necessary commands to instantiate the class and call its *Calculate()* method would be:

```
oMyClass = CREATEOBJECT( "TestCOM.SimpleDLL" )
lnResult = oMyClass.Calculate( 3256, 0.0575, "*" )
```

Note that when you build a DLL (or EXE) you can optionally include information about the build by using the Version button to set it up. Any information that is entered here will be available in the properties window for the DLL from Windows Explorer (see **Figure 6**).

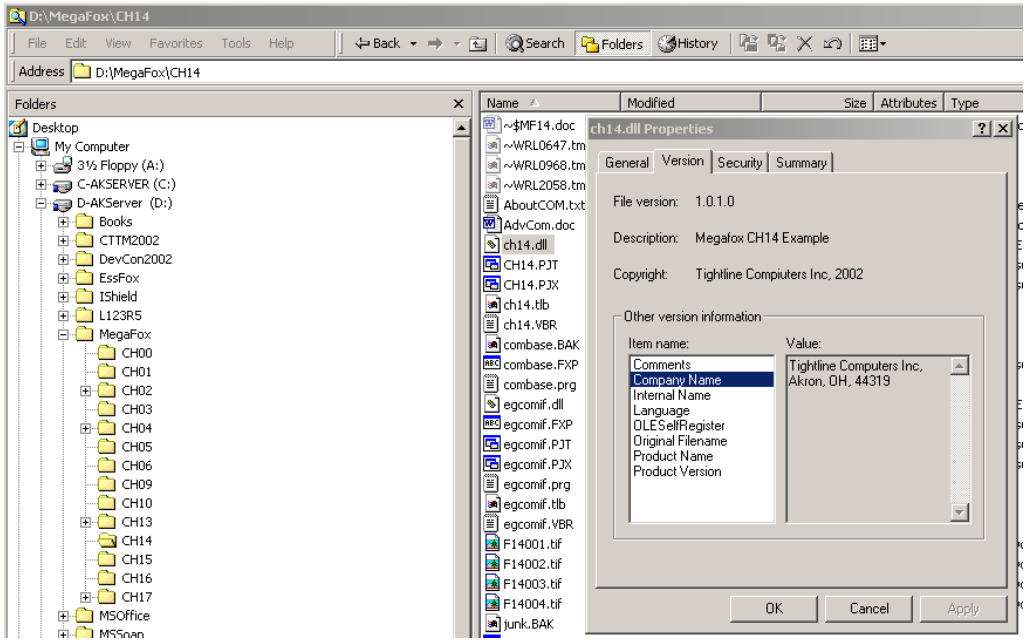


***Figure 6**. DLL Version information in Windows Explorer.*

Designing COM components (*Example: ComBase.prg*)

As we have already said, Visual FoxPro is a great tool for building COM components because it hides the complexity inherent in building a component that will work in a COM environment. However, there are still a number of issues that you, as the developer, need to take into account when designing and building components. While a full treatment of the subject is way beyond the scope of a single chapter in a book like this, there are two important issues that you need to consider when designing an object to work in a COM environment, namely:

- Error handling
- Interface implementation

Later in this section we will address each of these issues, but we'll begin by discussing the design of our "component root class" that we will use to create the classes that we use elsewhere in this chapter. (Note that the same basic class definition is also used in Chapter 16 to create the sample Web Service DLL.) This class is included in the sample code for this chapter as COMBASE.PRG.

The class is based on the Visual FoxPro Session base class and includes a set of generic properties and methods that we will need for all components as shown in **Table 2**.

***Table 2**. Component root class properties and methods.*

Name	Type	Description
nErrorCount	P	**Protected** property used to store the number of errors that have been logged.
aErrors	P	**Protected** property used to store error information in three columns. Converted by calling *GetErrors()* into an XML string that can be returned.
cErrTable	P	**Exposed** property that can be used to define the name of an error message table to be associated with the class.
lErrorMsg	P	**Protected** property used as a flag to indicate whether an error message table that has been defined is actually available.
Init()	M	**Exposed** initialization method. Calls the custom *EnvSet()* method and, if an error message table has been defined, also calls *OpenLocalTable()*.
Release()	M	**Exposed** method that calls the custom *CleanUp()* method.
GetErrors()	M	**Exposed** method that converts the contents of the internal error collection into an XML stream and returns it to the caller. Calling *GetErrors()* clears the error collection.
EnvSet()	M	**Protected** method, called, by default, from the *Init()* to set up the environment. Default behavior is to enable Multilocks and to disable the Bell and Error Logging.
CleanUp()	M	**Protected** method that calls *CloseLocalTables()* before releasing the current object.
Error()	M	**Protected** method that overrides the native Visual FoxPro error handler and logs errors to the internal error collection. See “How do I handle errors?” later in this chapter for more details.
LogError()	M	**Protected** method called from *Error()* to write error details to the internal collection.
GetItem()	M	**Protected** method to return a specified item from a delimited list. This is a generic function that is discussed in detail in Chapter 1 (“KiloFox Revisited”).
OpenLocalTable()	M	**Protected** method to open a table in the component’s environment and optionally set its buffer mode..
CloseLocalTables()	M	**Protected** method to close the error message table if one has been opened. Called from *CleanUp()*.
ValidateParam()	M	**Protected** method to check that a parameter is of the required data type.

The code in these methods, with the exception of the error handling, is simple enough to be self-explanatory. Note that of the properties and methods, only those custom items that are defined as “exposed” will appear in the Type Library. So the interface exposed by this class when built as a DLL appears as shown in **Figure 7**.

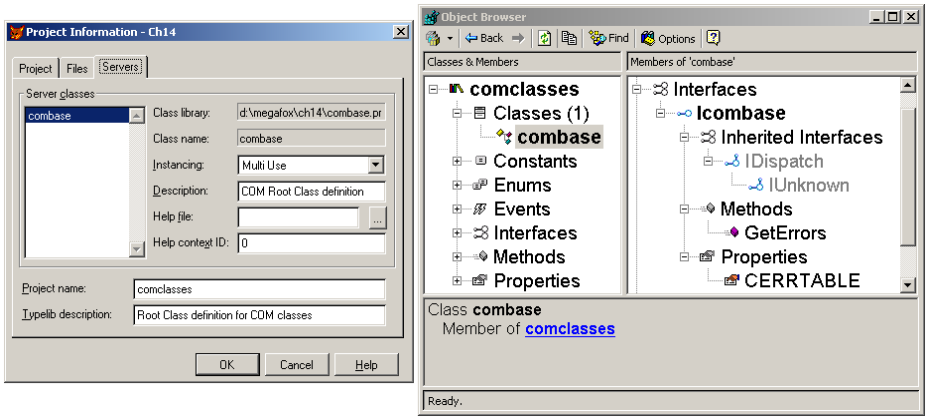


***Figure 7**. COM root class server information and interfaces.*

Note also that if the project (which is named CH14.PJX) is compiled and built, the resulting DLL is also named CH14.DLL by default. The name of the COM server inside that DLL will also be set to "CH14" by default. However, by defining the project name explicitly in the Servers tab of the Project Info dialog, we can force the DLL to be built with a server name that is independent of both the project and the physical file name. Figure 7 shows this very clearly. Our "CH14" project has been used to build a file named CH14.DLL, which contains a COM Server named *ComClasses* with a class named *ComBase*. Why do we mention this? Simply because this flexibility allows you to change the physical file name of the DLL (when you release a new version, for example) without having to change the name of either the server, or the classes, that it contains. This means that any existing code that uses the registered class name will simply get the version from whichever DLL was last registered on the machine. However, there is a catch to this!

It is precisely this behavior that gives rise to "DLL Hell." If a DLL that defines an older version of a specific class is registered *after* a newer version, all entries in the Registry will be updated to point to the newly registered, but older, version of the class. You can easily demonstrate this for yourself by adding a new method to the *ComBase* interface, building the project to a new DLL, and letting Visual FoxPro register it. When you create an object from the DLL in FoxPro you will see your new method appear in IntelliSense. Now use the Windows **RUN** command to re-register the original DLL (substitute the path and file names you have used as necessary) like this:

```
REGSVR32 D:\megafox\ch14\ch14.dll
```

If you now re-create the same object, using the same syntax, you will find that you have lost your new method and have, in fact, instantiated the older version of the DLL. The implication of this behavior is that if an application installs a specific version of a COM class that happens to be older than the existing version on the machine, code that was working perfectly one moment suddenly breaks. As mentioned earlier in the chapter, this problem has become so widespread, and has caused so much trouble, that the whole methodology has been changed in the .NET Framework.

How do I handle errors?

There are two basic methodologies that we want to discuss for dealing with errors. But irrespective of which approach we take, the first thing that we have to accept is that in a multi-threaded DLL we simply cannot allow an error to do the usual thing that Visual FoxPro does—display an on-screen message box! The reason is, of course, that we cannot know where such a DLL will be running, and should it be running on an unattended server, clearly having a modal message box displayed would be a problem.

For Automation servers, in which UI elements are permitted, the `SYS(2335)` function controls Visual FoxPro's "Unattended Mode". When enabled, any attempt to enter a modal state will raise an error. However, it is only necessary to manage this explicitly when creating out-of-process servers (that is, where StartMode = 2) because, for in-process servers, unattended mode is permanently enabled. So we can be sure that any attempt to generate a modal dialog will raise an error (Error #2031, "User-interface operation not allowed at this time"), but that doesn't actually take us any further to deciding how to deal with it.

Using COMRETURNERROR() *(Example STOPONERROR.PRG, DLLERROR.PRG)*

One way in which we can deal with an error is to ensure that whenever an error is encountered, the component immediately stops what it is doing and returns control to whatever object called it. The trick here is to ensure that the error information is available to the calling object after the component has stopped whatever it was doing when the error occurred. This is the role of the `COMRETURNERROR()` function that populates the COM exception structure with details of the error.

The COM exception structure can be accessed using the `AERROR()` function within Visual FoxPro to retrieve the information posted by the component. For such errors, the array is structured as shown in **Table 3**.

***Table 3**. COM error information.*

Element	Description
1	Visual FoxPro error number (1429)
2	Text of the Visual FoxPro error message
3	Fully qualified path and file name for the component
4	Text of the original error as raised by the component
5	Name of an associated Help file (empty if none)
6	Help context ID (0 if none)
7	The error number raised by the component

This mechanism can not only be used to trap any Visual FoxPro error, but also to raise custom exception errors using the `ERROR` command. The example class includes three custom methods that generate errors in various ways. The first, *GenError()*, generates a specific error and is coded as follows:

```
FUNCTION GenError( tnErrNum AS Integer ) AS VOID
  IF VARTYPE( tnErrNum ) = "N" AND NOT EMPTY( tnErrNum )
    lnErr = tnErrnum
  ELSE
```

```
      lnErr = 12
    ENDIF
    ERROR( lnErr )
  ENDFUNC
```

As you can see, this can be called explicitly with an error number, or simply allowed to raise a default "Variable Not Found" error. The second, *ShowMsg()*, attempts to display a message box, like this:

```
  FUNCTION ShowMsg As VOID
    MESSAGEBOX( "This is a modal dialog", 16, "Whoops!" )
    RETURN
  ENDFUNC
```

The last, *RaiseError()*, raises a custom exception by calling the **ERROR** command with the appropriate message. The specified text is attached to Visual FoxPro Error #1098 ("API function _UserError( ) was called").

```
  FUNCTION RaiseError()
    LOCAL lcErrString
    lcErrString = "An exception error was raised by " + _VFP.ServerName
    ERROR lcErrString
  ENDFUNC
```

The code in the *Error()* method simply creates the return message string using the standard parameters that are passed and calls **COMRETURNERROR()**, passing the message string as a parameter. Note the use of the *_VFP.ServerName* property to get the fully qualified path and file name of the DLL that is actually running. This has to be used because **SYS(16)** cannot be used in a DLL to correctly identify the source as it only returns the location of the runtime library:

```
  FUNCTION Error( tnError, tcMethod, tnLine )
    LOCAL lcErrStr
    lcErrStr = "Error " + TRANSFORM( tnError ) + ": " ;
             + MESSAGE() ;
             + " Occurred at Line " + TRANSFORM( tnLine ) ;
             + " of " + ALLTRIM( tcMethod )
    COMRETURNERROR( lcErrStr, _VFP.ServerName )
  ENDFUNC
```

The example program (DLLERROR.PRG) sets up an error handler that uses the **AERROR()** function to retrieve the error details and write them out to a text file for viewing. This program instantiates the class (which assumes that the project has been built as a file named ERRTEST.DLL) and then calls each of its methods in turn to generate the error in different ways as follows:

```
*************************************************************************
* Program....: DLLERROR.PRG
* Compiler...: Visual FoxPro 07.00.0000.9465
* Purpose....: Illustrate the use of ComReturnError()
*************************************************************************
```

```
LOCAL loDll
ON ERROR DO errproc
*** Clewar any old error log
IF FILE( 'errors.txt' )
  DELETE FILE errors.txt
ENDIF

*** Instantiate the DLL which is built as "errtest"
loDll = CREATEOBJECT( 'errtest.stoponerr' )
*** Try the MessageBox first...
loDll.ShowMsg()
*** Then the GenError
loDll.ShowMsg( 1214 )
*** ANd finally the custom Error
loDll.RaiseError()
ON ERROR
MODIFY FILE errors.txt NOWAIT
RETURN

FUNCTION ErrProc
LOCAL lnErrs, lnCnt, lcStr
LOCAL ARRAY laErr[1]
*** Get the details into an array
lnErrs = AERROR( laErr )
*** Write errors out to file
lcStr = ""
FOR lnCnt = 1 TO 7
  lcStr = lcStr + "[ Element " + TRANSFORM( lnCnt ) + "] = " +;
            TRANSFORM( laErr[ lnCnt] ) + CHR(13) + CHR(10)
NEXT
*** Add a blank line to separate groups
lcStr = lcStr  + CHR(13) + CHR(10)
STRTOFILE( lcStr, 'errors.txt', 1 )
RETURN
```

Note that although the component stops executing code as soon as an error is encountered, it is not unloaded from memory and remains available without needing to be re-instantiated.

Using error logging *(Example LOGERROR.PRG, DLLELOG.PRG)*

An alternative to using the **COMRETURNERROR()** function is to log errors internally in the component and return all the error information to the client on completion of processing. The benefit of this approach is that instead of stopping at the first error, processing continues thereby allowing multiple errors to be recorded and returned to the caller in one single operation. Admittedly there are times when you may want to stop at the first error, but in other scenarios this may be a very desirable solution—for example, when validating data. This approach can also be used to simplify the otherwise tricky task of debugging in a COM component.

The sample project *LogOnErr* contains a subclass of our standard COM class named *LogOnError*, which includes the same three methods as were used in the preceding example to generate errors. A single exposed method, named *TestLogging()*, calls each of these methods in turn to raise the errors:

```
FUNCTION TestLogging()
LOCAL lcRetStr
  WITH This
    *** Call the various methods that raise the errors
    .RaiseError()
    .GenError( 45 )
    .ShowMsg()
    *** Now get the errors into a string to return them
    lcRetStr = IIF( .nErrorCount > 0, .GetErrors(), "" )
    RETURN lcRetStr
  ENDWITH
ENDFUNC
```

The default behavior of our root class is to handle errors by logging them to an internal array and updating a count property. The *GetErrors()* method can be called to retrieve any messages stored in that array as an XML stream and clear the array and counter. The example program, DLLELOG.PRG, instantiates the DLL, calls the test method, and writes the results out to a text file (see **Figure 8**).

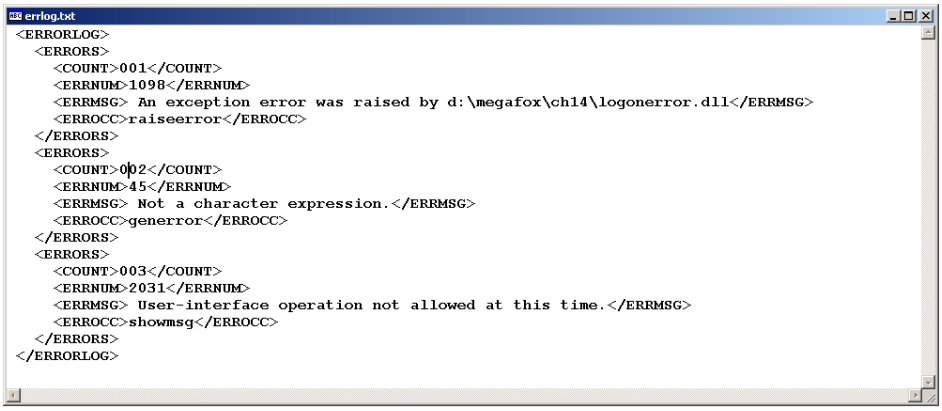


***Figure 8**. Returning the error log from a component.*

The sample code included with this chapter defines the format of the XML file that we like to use, but there is, of course, no particular reason why you should adhere to this format, or indeed to the structure of the array that we use.

One additional benefit of this approach is that the custom *LogError()* method can be called explicitly to report program status information. This is shown by the custom *ProcessReporting()* method, which simply writes a "Status" error to the errors array and looks like this:

```
FUNCTION ProcessReporting()
LOCAL lnCnt, lcRetStr
  WITH This
    *** Run a "10-Step" process
    FOR lnCnt = 1 TO 10
      .LogError( "Status", "Processing Step " + TRANSFORM( lnCnt ), ;
                                          _VFP.ServerName ) + PROGRAM())
    NEXT
```

```
      *** Now get the "errors" into a string to return them
      lcRetStr = IIF( .nErrorCount > 0, .GetErrors(), "" )
      RETURN lcRetStr
    ENDWITH
  ENDFUNC
```

Figure 9 shows the XML produced by the *ProcessReporting()* method.

repstatus.txt

```
<ERRORLOG>
  <ERRORS>
    <COUNT>001</COUNT>
    <ERRNUM>Status</ERRNUM>
    <ERRMSG> Processing Step 1</ERRMSG>
    <ERROCC>d:\megafox\ch14\logonerror.dll.PROCESSREPORTING</ERROCC>
  </ERRORS>
  <ERRORS>
    <COUNT>002</COUNT>
    <ERRNUM>Status</ERRNUM>
    <ERRMSG> Processing Step 2</ERRMSG>
    <ERROCC>d:\megafox\ch14\logonerror.dll.PROCESSREPORTING</ERROCC>
  </ERRORS>
  <ERRORS>
    <COUNT>003</COUNT>
```

***Figure 9**. Using the logging process for status reporting.*

> ! *This technique does allow us to address a minor problem that arises when working with COM components, that of debugging code. When a component is running it cannot be debugged in the usual way, and even though the code may function perfectly when run directly in Visual FoxPro as a class, there is no guarantee that the same code will run correctly when compiled into a DLL. This is especially true when paths are involved because some of the usual Visual FoxPro functions return different values (for example, PROGRAM()) when called from within a DLL. Unfortunately, there are no hard and fast rules that we can offer; the best advice is just to try it!*

How do I implement an interface?

The section "All about interfaces" earlier in this chapter described how COM interfaces are all based, ultimately, on the fundamental *IDispatch* and *IUnknown* interfaces and the significance that these methods have for Visual FoxPro. However, although we illustrated how to define an interface, we did not actually discuss how you can make use of interfaces to create your own components.

The key to working with COM interfaces is to remember that the purpose of the interface is not to define the full functionality of the object, but merely to define how other objects can interact with it. In other words, the interface defines *usage*, not *implementation*. It follows, therefore, that only those PEMs that are intended to be accessed, or manipulated, by other objects should be exposed as part of an interface. It is an absolute rule that when an object implements an interface, it must implement that interface exactly as it was originally defined. In fact, when you try to compile a class that implements an interface, Visual FoxPro will

check the Type Library to ensure that the definitions match exactly and will generate errors for any deviation.

This behavior has a couple of important consequences. First (and most obviously), it ensures that all components that implement a specific interface implement it in exactly the same way. This is clearly important, as things would get very confusing if you had to call a "standard" method differently depending on the object that was implementing it. Second (and perhaps less obviously), it allows us to define a set of standardized interfaces that we can then reuse in our own components. As noted elsewhere, Visual FoxPro already contains several good examples of implementing COM and, since we see no point in duplicating information that is already available elsewhere, we shall illustrate how to define and implement a set of standardized interfaces for use when creating custom components.

Defining the interfaces (*Example: ComIF.pjx; ComIF.prg*)

The COMIF.PJX project, included with the sample code for this chapter, consists of a single PRG file (also named COMIF) that defines interfaces for two classes, as follows:

```
*********************************************************************
* Program....: COMIF.PRG
* Compiler...: Visual FoxPro 07.00.0000.9465
* Purpose....: Define a set of standard COM Interfaces
*********************************************************************
* ...........: xMsgHandler :: Defines Interface for Message Handler
*********************************************************************
DEFINE CLASS xMsgHandler AS session OLEPUBLIC
 *** Property to hold default title for Message Displays
 cDefTitle = ""
 *** Use COMATTRIB to define property characteristics
 DIMENSION cDefTitle_COMATTRIB[ 5 ]
 cDefTitle_COMATTRIB[ 1 ] = 0
 cDefTitle_COMATTRIB[ 2 ] = "Default Title for use when none specified"
 cDefTitle_COMATTRIB[ 3 ] = "cDefTitle"
 cDefTitle_COMATTRIB[ 4 ] = "String"
 cDefTitle_COMATTRIB[ 5 ] = 0

 FUNCTION ShoMsg( tcMsgTxt AS String, tnStyle AS Integer, tcTitle AS String );
    AS Integer ;
    HELPSTRING "Handles generation of a message in the appropriate format"
 ENDFUNC

 FUNCTION GetMsg( tnMsgID AS Integer, tcTable AS String ) AS String ;
   HELPSTRING "Returns text associated with passed ID from message table"
 ENDFUNC
ENDDEFINE

*********************************************************************
* ...........: xDataFinder :: Defines Interfaces for SEEK() and LOCATE
*********************************************************************
DEFINE CLASS xDataSearch AS session OLEPUBLIC
 FUNCTION IdxSch( tuFindVal AS Variant, tcTable AS String, tcTag AS String );
   AS Integer ;
   HELPSTRING "Perform index search on passed table and return Record Number"
 ENDFUNC
```

```
 FUNCTION FldSch( tuFindVal AS Variant, tcTable AS String, tcFld AS String ) ;
   AS Integer ;
   HELPSTRING "Carry out a search on the table and return a Record Number"
 ENDFUNC
ENDDEFINE
```

Note that the *xMsgHandler* class includes a custom property in its interface and uses the new **COMATTRIB** array to declare specific settings for it. Each class then defines two methods using the enhanced type declarations for parameters and return values. When built into a DLL and examined in the Object Browser, the result is shown in **Figure 10**.

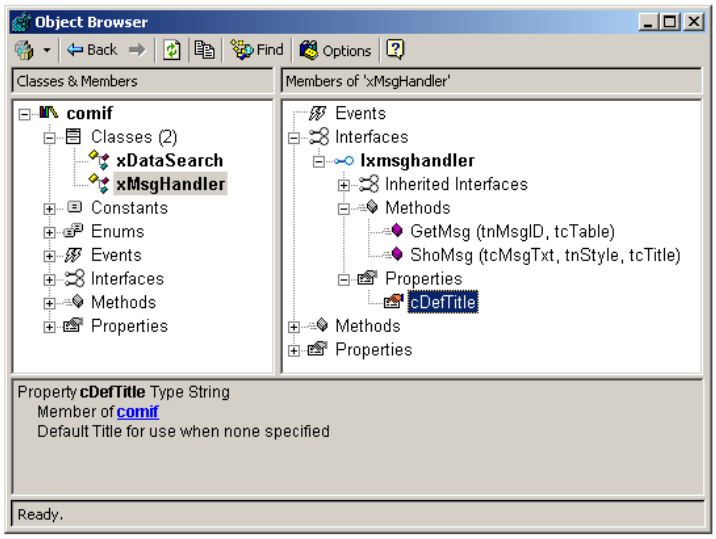


***Figure 10**. COM interface definitions.*

Implementing the interfaces

In order to use these definitions, all that is necessary is to define a new class that implements either, or both of them. Remember, all that can ever be inherited from a COM interface is the definition (in this case the classes have no code anyway, but even if they did it would not be inherited). The example program defines a new class that implements both of our new interfaces by including the **IMPLEMENTS** keyword in the **DEFINE CLASS** statement. The full syntax for **IMPLEMENTS** is:

```
IMPLEMENTS <Interface> [EXCLUDE] IN <FileName>| <ClasID+Version> | <ProgID>
```

where the optional **EXCLUDE** keyword ensures that the specified interface is not visible in the type library when the component is built. The location of the interface can be defined in any of three ways as follows:

- By explicitly specifying the location of the Type Library. This is not really a very good method when the source code for components is going to be distributed because

the full path has to be specified and that may not always be the same. For example:

```
IMPLEMENTS xMsgHandler IN "D:\MEGAFOX\CH14\COMIF.DLL"
```

Of course, if only the DLL is being distributed, the reference is resolved at compile time so this is not an issue.

- By supplying the Registry ClassID, together with the relevant version information. This information can be gleaned from the VBR file that is generated when the component is compiled. The main benefit (and the potential problem) with this method is that it does make the implementation version-specific. For example, when we compiled our example, the ClassID generated was:

```
HKEY_CLASSES_ROOT\TypeLib\{9AEA3A48-AB62-4233-AA20-9F0B3951CAD6}\1.0 =
comif Type Library
```

which would be coded as follows:

```
IMPLEMENTS xMsgHandler IN {9AEA3A48-AB62-4233-AA20-9F0B3951CAD6}\1.0
```

- By supplying the version independent ProgID for the source. This can be gleaned from the VBR file, but you can also get it by instantiating the class directly and using option 2 of the native Visual FoxPro `COMCLASSINFO()` function. The benefit of this method is that the version independent ProgID does not change with different versions, and always returns a reference to the last version that was installed on the local machine (beware "DLL Hell"). For example:

```
IMPLEMENTS xMsgHandler IN "comif.xMsgHandler"
```

The easiest way to get the interface definitions right when implementing them is to open the original class in the Object Browser and then drag and drop the required interface from the left hand pane of the browser directly into your program file. Visual FoxPro will automatically duplicate the entire class definition for you, and this ensures that nothing is missed out or misspelled. Of course, if you really like hard work, you can simply re-type everything yourself too. The only snag with the drag-and-drop approach is that in addition to the interface definitions, a full class definition is added by default (named *myclass*), but that is easy enough to remove. **Figure 11** shows the sequence of events to create a new class definition that implements both of our defined interfaces using the drag-and-drop technique. There are, however, a couple of things to note.

First, notice that all method names are prefixed with the name of the interface from which they derive. This both ensures uniqueness when implementing multiple interfaces that may contain the same method names, and also provides the necessary information for checking the source definition. For this reason, it is usual to either protect implemented interface methods, or explicitly exclude them from the type library and instead provide a set of exposed method names that call the inherited methods and properties internally.

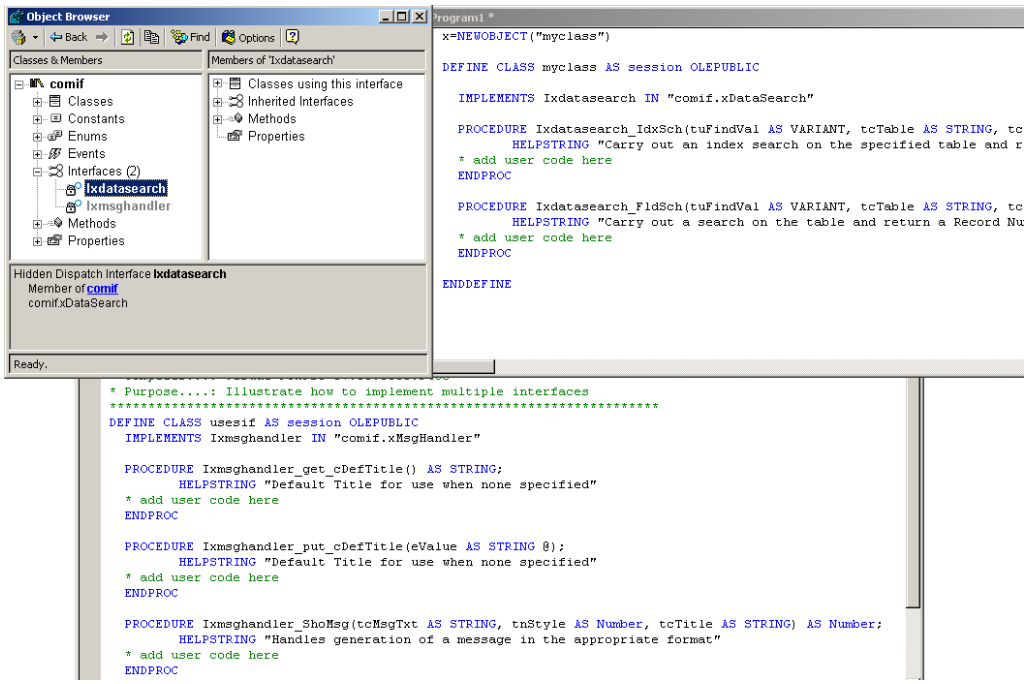


***Figure 11**. Drag-and-drop interface definitions from the Object Browser.*

Second, notice that properties are actually implemented in COM with two methods, a *Get()* method to retrieve the value, and a *Put()* method to set it. You can see these in the lower part of Figure 11 where they define the *cDefTitle* property in the *xMsgHandler* interface. If you choose to code an implemented interface directly, don't forget to include both these methods for each property.

The main benefit of defining interfaces in this way (apart from ensuring adherence to the specific interfaces) is that a single component can inherit as many, or as few, interfaces as it needs. Once the necessary interfaces have been defined, all that remains to be done is to add the necessary implementation code. For an example of how this is done in practice, see the *VFPSAXHandler* class example in Chapter 17.

And there's more!

In addition to those elements that we have covered, there are several advanced features including COM Event Binding and a new set of functions to discover and control Automation server invocation mode (**`SYS(2334)`**), critical sections support (**`SYS(2336)`**), and Windows NT Service support (**`SYS(2340)`**). Four more new functions (**`SYS(3095)`**, **`SYS(3096)`**, **`SYS(3097)`**, and **`SYS(3098)`**) are specifically concerned with accessing the *IDispatch* and *IUnknown* interfaces, and while these are unlikely to be used in everyday Visual FoxPro development there are some very specific situations that will require their use (for example, when calling API routines that require an object pointer).

How can I use COM in the real world? (*Example: SecCom.pjx*)

The purpose of this little example is to show how you can create a COM component in Visual FoxPro to handle the validation of user logins. While not exactly rocket science, this little component shows the benefit of using a data aware tool like Visual FoxPro for developing components to run in the middle tier. The project is stored in its own subdirectory and uses a Visual FoxPro database whose structure is illustrated in **Figure 12**.

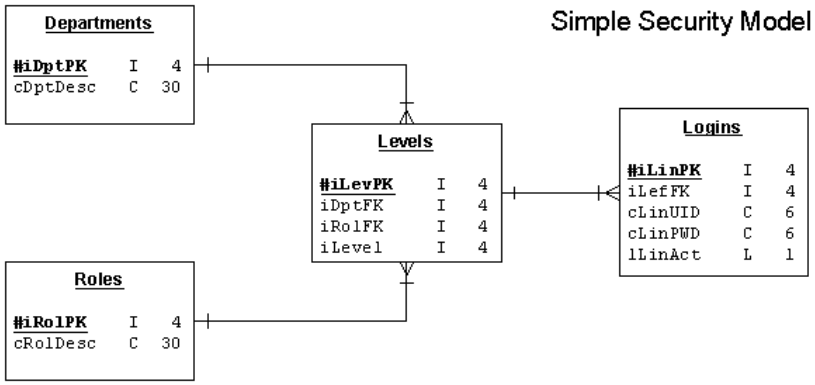


***Figure 12**. SecCOM database structure.*

This structure allows us to define an access level structure for users at the intersection of Department and Role as illustrated by **Table 4**. The levels are defined as allowing various combinations of Read, Edit, Add, and Delete functionality as shown in the table.

***Table 4**. SecCOM security model.*

Roles	Ops department	IT department	Sales department
Director	20 (Read/Edit)	40 (Read/Edit/Add/Delete)	10 (Read Only)
Manager	40 (Read/Edit/Add/Delete)	40 (Read/Edit/Add/Delete)	30 (Read/Edit/Add)
Senior Staff	30 (Read/Edit/Add)	40 (Read/Edit/Add/Delete)	20 (Read/Edit)
Junior Staff	20 (Read/Edit)	30 (Read/Edit/Add)	10 (Read Only)
Guest/Temporary	10 (Read Only)	10 (Read Only)	10 (Read Only)

One of the main benefits of doing things this way is that the Visual FoxPro tables can be updated at any time. This means that it is possible to data drive your components, and that is a very attractive proposition indeed in the context of Web servers and "24x7" applications. By data driving a component we make it possible for modifications to be made to the system without the necessity of taking it offline. For an example of data driving Web page generation, see the *Lister* and *Render* classes in Chapter 16.

Building the component

The actual code in our component is reasonably straightforward. The *SecCom* class is defined in SECCOM.PRG as a subclass of our custom *ComBase* class. It defines an *Init()* method that calls the root class method and then calls the custom *Setup()* method to open the tables locally.

```
PROTECTED FUNCTION Setup()
  LOCAL ARRAY laTables[4,1]
  LOCAL llRetVal, lnCnt, lcLoc
  WITH This
    *** Get our current location into a variable too
    lcLoc = _VFP.servername
    *** Set up the table array
    laTables[1,1] = ADDBS( JUSTPATH( lcLoc )) + 'departments'
    laTables[2,1] = ADDBS( JUSTPATH( lcLoc )) + 'roles'
    laTables[3,1] = ADDBS( JUSTPATH( lcLoc )) + 'levels'
    laTables[4,1] = ADDBS( JUSTPATH( lcLoc )) + 'logins'
    FOR lnCnt = 1 TO ALEN( laTables, 1 )
      *** Try and open the table
      IF NOT .OpenLocalTable( laTables[ lnCnt, 1 ] )
        *** Log the error if we fail
        .LogError( 'Setup', 'Unable to open table: ' ;
                            + laTables[ lnCnt, 1 ], lcLoc )
      ENDIF
    NEXT
    *** We also want SET EXACT ON - this is a Private DataSession
    SET EXACT ON
    RETURN
  ENDWITH
ENDFUNC
```

Note that even though our data is in the same physical directory as the DLL, we still need to explicitly pass the fully qualified location of the tables. This is so that when the DLL is instantiated from another location the paths are set up correctly and the data can be found. The only problem is that the usual mechanisms for retrieving the path (for instance, `SYS(5) + CURDIR()`, `SYS(16)`, or `HOME()`) fail to give us the correct information when used inside a DLL. The solution, as shown in the preceding code, is to use `JUSTPATH()` to retrieve the physical path from the *ServerName* property of the Visual FoxPro Application object to determine the physical location of the DLL at run time.

A single exposed method, named *GetUserLevel()*, is where the actual work is done, and this method expects to receive two parameters, a user ID and a password. The first part of the method ensures that both parameters are at least of the expected data type and if either is not, returns an error string in XML format by calling the root class *GetErrors()* method:

```
FUNCTION GetUserLevel( tcUID AS String, tcPWD AS String ) AS String
LOCAL lcUID, lcPWD, lnLevFK, lcAccess
  WITH This
    *** Check the parameters
    IF NOT .ValidateParam( tcUID, "C" )
      .LogError( "GetUserLevel", "Invalid User ID: " + TRANSFORM( tcUID ), ;
                                                          _VFP.ServerName )
    ELSE
      *** Format Login - not case sensitive!
      lcUID = UPPER(ALLTRIM( tcUID ))
    ENDIF
    IF NOT .ValidateParam( tcPWD, "C" )
      .LogError( "GetUserLevel", "Invalid Password: " + TRANSFORM( tcPWD ), ;
                                                          _VFP.ServerName )
    ELSE
      *** Password IS case sensitive, just trim it
      lcPWD = ALLTRIM( tcPWD )
```

```
      ENDIF
      *** If the parameter check failed abort here
      IF This.nErrorCount > 0
        RETURN .GetErrors()
      ENDIF
```

Assuming that the parameters are valid, the remainder of the code in this method is straightforward Visual FoxPro code that first does a **SEEK()** in the Logins table. If the specified user ID is found, the passed-in password is checked against that in the table. Finally, we check the setting of the flag that denotes a currently active user and, if all is well, return the access level assigned to that user from the Levels table. Failure at any point returns an access level of "0".

```
      *** See if we have this particular Login/Password combination
      *** Start by setting the Level ID to 0
      lnLevFK = 0
      *** Does the login exist?
      IF SEEK( lcUID, 'logins', 'clinuid' )
        *** If so, is the password correct?
        IF lcPWD == ALLTRIM( logins.clinpwd )
          *** And finally is this still an active user
          lnLevFK = IIF( logins.llinact, logins.ilevfk, 0 )
        ENDIF
      ENDIF
      *** If lnLevFK > 0 then we have a valid user
      lcAccess = "0"
      IF lnLevFK > 0
        IF SEEK( lnLevFk, 'levels', 'ilevpk' )
          lcAccess = TRANSFORM( levels.iLevel )
        ENDIF
      ENDIF
      RETURN lcAccess
    ENDWITH
  ENDFUNC
```

Notice that the return value from this method is always in character format. This is no accident; it is very definitely the responsibility of the client object to interpret the value that the component returns. The component cannot possibly know how, or from where, it is being called and therefore cannot afford to make assumptions about the capabilities of its client. By the same token, it is reasonable to assume that any client capable of calling the component will also be capable of handling a character string.

Testing the component in Visual FoxPro

Having built the project into a DLL, all that remains is to test it. This can be done directly in Visual FoxPro by instantiating the object from the command line as usual:

```
oT = CREATEOBJECT( "seccom.seccom" )
lcLevel = oT.GetUserLevel( "AaAaAa", "aaaaaa" )
```

The result should be a value of 40, that being the level assigned to the user with the specified ID and password. Altering user ID and password parameters allows us to ensure that the component is functioning as intended. (Obviously a real component would have more than

this single method, but for the purpose of this example, one method will suffice.) We are now ready to deploy our component in a new environment.

Testing the component with ASP

In order to test our wonderful and powerful component, we can set up an Active Server Page that can run on our local machine. In order to do this, three things have to be done first, as follows:

- Start the Internet Information Services manager and create a new virtual directory that points to the directory in which your DLL resides (our default is, for example, D:\MEGAFOX\CH14\SECCOM, and we also named the virtual directory "SecCom").
- In the physical directory pointed to by the virtual directory, create a new text file named DEFAULT.ASP that will become our simple log-in page.
- In the physical directory pointed to by the virtual directory, create a new text file named SECCOM.ASP that will become the page from which we actually call our component.

Creating Default.ASP

The DEFAULT.ASP page (or DEFAULT.HTM) is the first thing that is run whenever you navigate to a Web site (or, in this case, the virtual directory). It can do whatever you want it to, but for the purpose of testing this component it is going to present a couple of textboxes that can be filled in with the user ID and password. **Figure 13** shows the page as it appears in the browser when you navigate to the virtual directory (http://localhost/seccom, or whatever name you assigned).

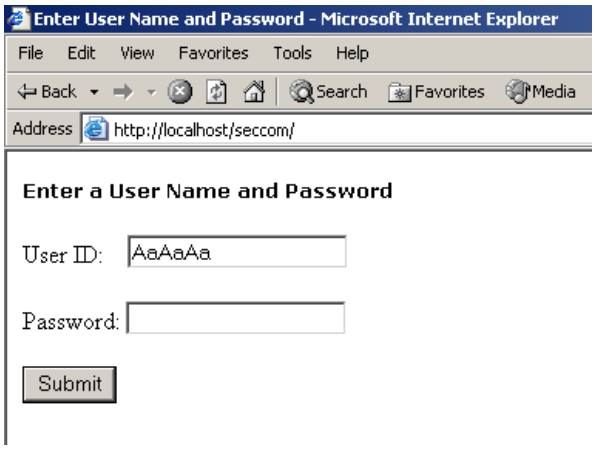


***Figure 13**. Testing the access component in ASP.*

The code in DEFAULT.ASP defines the title for the page, and sets up the text boxes and the Submit button. It also identifies, in the action attribute of the form tag, the page to call (in this case, SECCOM.ASP).

```
<%@Language="VBScript"%>
<%
' Title:       Default.asp
' Subtitle: This is the main screen FOR MegaFox Chapter 14 Sample dll call
%>
<%
  Response.Buffer = True
  Response.ExpiresAbsolute = Now() - 1
  Response.Expires = 0
  Response.CacheControl = "no-cache"
%>
<html>
<head>
<title>Enter User Name and Password</title>
</head>
<body>
<form name="Login" method="POST" action="SecCom.asp">
<font face="Verdana, Arial, Helvetica, sans-serif" size="2"><b>Enter a User
Name and Password</b></font>
<p>User ID:    <input type="text" name="txtUName" size="20"></p>
<p>Password: <input type="password" name="txtPwd" size="20"></p>
<p><input type="submit" value="Submit" name="cmdSubmit"></p>
</form>
</body>
</html>:
```

Creating the custom ASP page

The code in the custom page (SECCOM.ASP) retrieves the values from the form, instantiates the component, and then calls its *GetUserLevel()* method passing the retrieved values as parameters. The result, in this case, is simply returned as a string and displayed as a new page:

```
<% @ Language=VBScript %>
<%
'    Title:       SecCom.asp
'    Subtitle:    This calls the VFP com object and returns the access level
%>
<%
cUName = Request.Form( "txtUName" )
cPWD = Request.Form( "txtPwd" )
Set oChkAccess = CreateObject( "seccom.seccom" )
cRetVal = oChkAccess.GetUserLevel( cUName, cPwd )

IF LEN(cRetVal) > 2 THEN
    cHTML = "An error occurred: " & cRetVal
ELSE
    cHTML = "Access Level = " & cRetVal
END IF

Response.Write( cHTML )
%>
```

This is really all that there is to it and, while this is an extremely simple example, it does show how terribly easy it is to create COM components using Visual FoxPro. The same component could, of course, just as easily be called by anything that is capable of instantiating a COM class.

How do I distribute a component?

The whole topic of distribution is covered in more detail in Chapter 11 ("Deployment"). The process for distributing a component (whether built as an EXE or a DLL) is actually little different from distributing any other type of Visual FoxPro application. The main difference is in the runtime files that you need to include, and that depends upon the type of component. You need either:

VFP7R.DLL (for single-threaded DLL, EXE, or APP files)

or

VFP7T.DLL (for multi-threaded DLLs)

You also need to ensure that the language-specific resource file is included. This is named "VFP7XXX.DLL" where "xxx" identifies the language. Thus, "VFP7ENU.DLL" is the English language version, while "VFP7KOR.DLL" is the Korean language version.

Note that when distributing a component it must also be registered on the target machine. All standard distribution tools include functionality to register components as part of an installation process, but the exact mechanism for specifying it varies from tool to tool.

How do I register a component on my machine? (*Example: Register.prg*)

Actually, you don't need to bother! Visual FoxPro very kindly registers a component (using the information that is written to the VBR file) on your machine whenever you compile it. The only snag is that if, during development, you decide to change the name, or the location, of your component, Visual FoxPro just re-registers it as a "new" component the next time you build it. This means that you can very easily end up with orphan references in your Registry—never a good idea! It really is worthwhile to make sure that existing references are removed before re-building an existing component.

Of course, you will immediately be thinking that if this is something that needs to be done before a build, then a project hook is a good solution, and you are absolutely correct. You could make use of the *BeforeBuild* event to automatically un-register an existing DLL before it is rebuilt. However, if this is something that you really don't do very often, you may prefer to do it explicitly so that it is only done when really needed. Whichever option you adopt, the same command is used. It can be executed from within a Visual FoxPro method, or program, using the native **RUN** command or (if you prefer to be Win2K-compliant) the Windows API **SHELLEXECUTE()** function, or even just executed directly using the Windows Run dialog from the system Start menu.

Whichever method you use, you will need to use the correct command depending on whether it is a DLL that you are dealing with or an EXE. Alternatively you can use the *Register()* function, included in the download code for this chapter, which handles the process transparently and provides for either programmatic or interactive use.

Registering DLLs using Regsvr32

REGSVR32.EXE is a standard Windows utility whose function is to register and un-register DLL and ActiveX files that are self-registering. All DLLs created by Visual FoxPro are self registerable, which means that they support two standard API functions named

DLLRegisterServer() and **DLLUnRegisterServer()**. In order to register a file, simply pass the fully qualified path and file name as a parameter, like this:

```
REGSVR32 D:\megafox\ch14\seccom\seccom.dll
```

To un-register one, simply add the /u switch like this:

```
REGSVR32 /u D:\megafox\ch14\seccom\seccom.dll
```

By default RegSvr32 displays a dialog indicating the success of the requested operation (see **Figure 14**), but an optional /s command line parameter can be used to force the utility to run in "silent" mode and suppress the display of these dialogs. This is used when it is necessary to run RegSvr32 programmatically.

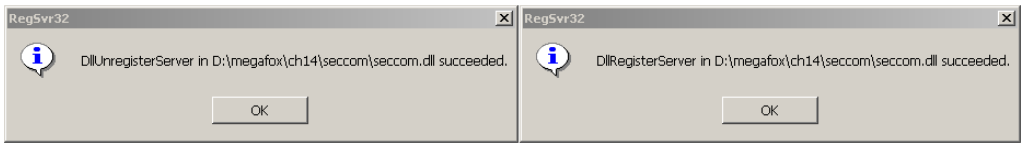


***Figure 14**. RegSvr32 dialogs.*

Registering EXEs

Out-of-process servers that are compiled as EXE files must also be registered and, like DLLs, those built by Visual FoxPro are self-registering. However, since EXE files are, by definition, executable, there is no need to use an external command to initiate the registration. The relevant functions are already part of the EXE, and all that is necessary is to call the EXE itself and specify the appropriate command line switch—either /regserver to register the file, or /unregserver to un-register it.

Using our "register" function to handle registration

The sample code for this chapter includes a wrapper program (REGISTER.PRG) that can be used to explicitly register or un-register DLL or EXE files by calling the **SHELLEXECUTE()** API function to run the appropriate command. The program accepts up to three parameters as follows:

- ***tlUnRegister***—Passing any non-empty value as the first parameter sets the program's action flag to UN-REGISTER. The default behavior, when no parameter is passed, is to attempt to REGISTER the specified file.
- ***tcFile***—The second parameter is the file to register or un-register. A fully qualified path and file name is required and, if nothing is passed, the **GETFILE()** function is called to enable a file to be specified.
- ***tlNoShow***—The third parameter can be used to suppress the error message display when calling the function programmatically

The result is that in order to register a file interactively, all that is needed is to call the program with no parameters and just pick the file:

```
? Register()
```

To un-register a file interactively just pass any non-empty value:

```
? Register( .T. )
```

The function passes back the value that the **SHELLEXECUTE()** function returns. This will be an integer, and a value of 33 or higher indicates that the operation succeeded. By default a message box is displayed when the function fails that lists the possible errors (see **Figure 15**).

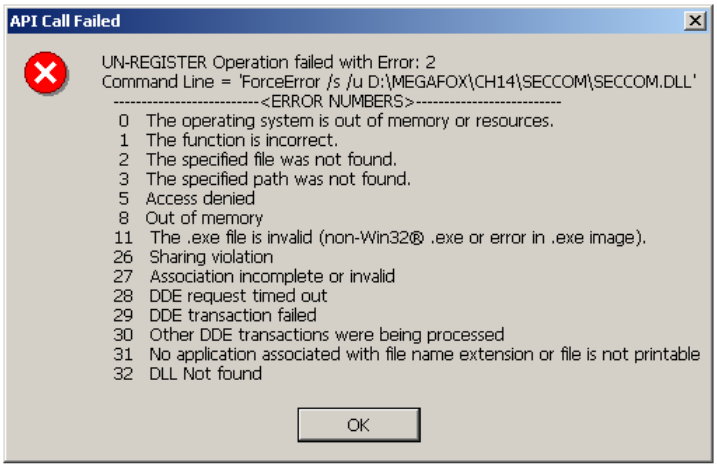


***Figure 15**. REGISTER.PRG error report.*

However, the program can also be called programmatically and, providing that a valid file name is passed, will run without user intervention. In this scenario you must explicitly pass the third parameter as True to suppress the display of the error message box as follows:

```
lnResult = Register( .F., "D:\MEGAFOX\CH14\SECCOM\SECCOM.DLL", .T.)
```

Conclusion

A full treatment of all of the capabilities and potential that Visual FoxPro offers in the context of COM and COM+ development would require far more than a single chapter. However, we have made reference several times in the course of this chapter to the extensive coverage given in the Visual FoxPro Help and Sample files to COM features, and you will find an example in Chapter 16 (“VFP on the Web”) of using a Visual FoxPro component. You can find out more information on using COM in Visual FoxPro by downloading the set of excellent white papers on the topic from the Visual FoxPro Web site at http://msdn.microsoft.com/vfoxpro/.

The new COM features introduced in Version 7.0 have made Visual FoxPro an even better tool with which to build components, and when taken together with its superior XML handling and integrated support for XML Web Services, it is clear that Visual FoxPro now merits very serious consideration as a tool for developing COM components that can be called from a variety of applications, whether to provide services or simply to gain access to data.

Chapter 15
Designing for Extensibility

The principal focus of this book is on extending the functionality of Visual FoxPro applications by making use of other tools and products. While we have given a lot of specific examples of how to do things, we have not really addressed the issues associated with designing an application so that it is extensible to begin with. This chapter attempts to correct that omission by discussing some of the basic design concepts and patterns and illustrating how they can be implemented in a Visual FoxPro environment.

How do I design an application?

The short answer to that question is "however you want to" or, if you prefer, the more usual Visual FoxPro answer, "it depends…". However, in order to make any sense of either answer we need to step back and look at the various ways in which an application *could be* designed. Probably the simplest type of application (which most, if not all, of us started out doing) can be defined as the "monolithic" application.

Monolithic applications

Typically, applications built in this way are not actually designed, they "evolve"—usually from a very simple model. For example, suppose we want to build an application to keep track of the Christmas cards we send every year. We might well start out with a single table (for the names and addresses), a single form to maintain that table, and one report so that we can print the list of people to send cards to. All of the necessary code can be placed in the standard methods of objects on the form (or perhaps in custom methods added to the form) and, to be honest, nothing more is really necessary.

However, after Christmas we suddenly realize that we could use the same basic application to keep track of people's birthdays too. Of course, we'd need to add an extra field to store the birth dates. The killer word here is *add*, because it indicates that our application is starting to evolve! Still, adding an extra field for a birthday is no big deal. But wait a moment, we should also consider that we send Christmas cards to lots of people whose birthdays we don't know and, even if we did we probably wouldn't send them cards anyway (business contacts, for example). So it's not just a single field that we need; we also need to start *classifying* the entries into those who get Christmas cards only, and those who get both Christmas and birthday cards.

However, this raises another problem! We generally send one Christmas card to each "family" on our list, but even if we were to send individual cards to different members of the same family it wouldn't really matter because Christmas is always on the same date for everyone. That does not apply to birthdays. For example, although Andy's sister and brother-in-law live at the same address, they have different birth dates. So do their children (Andy's niece and nephew), so for that one family we need to track at least *four* different birthdays. In fact, now that we think about it, we really need much more than our single table to support this functionality. Of course, changing the data model means that our original single form will

have to be changed radically and, before we know it, we are re-writing our application from the ground up.

This trivial example illustrates both the life cycle and the problems associated with monolithic applications. They evolve because there is always something extra that can be done by just *adding* a little to what is there. The problems arise because the data, the processing, and the user interface are inextricably linked, making the application inflexible and making it difficult, if not actually impossible, to "add" that one little item without making extensive changes. The solution, of course, is not to do it like this to begin with.

Layered applications

So, if monolithic applications are not the answer, what is? The solution is inherent in the fact that even the very simple application we just described has three quite independent constituents, which can be visualized as occupying separate layers in the application:

- **Data**—We always need a mechanism for storing the data that an application requires in order to carry out its function. Typically this will involve one or more tables in a database.
- **Processing**—We need processing to translate the *data* that is stored in our tables into the *information* that is the ultimate product of an application.
- **Interface**—We need some form of interface in order to be able to interact with its users. It does not matter whether those users are human beings or other systems, we always need a defined interface.

Is this the three-tier model?

Well, sort of. Strictly speaking the three-tier model is an architecture for implementing a layered design. In it we allocate each of the constituent parts of an application to a separate "tier" (see **Figure 1**), which is responsible for the delivery of the appropriate set of services.

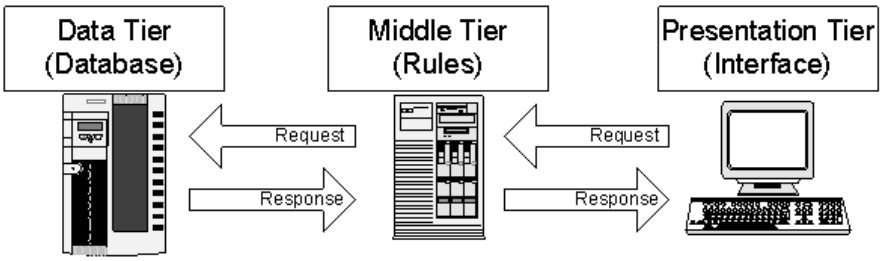


***Figure 1**. Basic three-tier model.*

It is important to recognize that the three-tier model is really describing a hardware architecture rather than a software design. The basic concept is that each tier resides on a different physical machine. Thus the data tier is provided by a database application running on a dedicated "data server," the middle tier by a rules application running on an "application server," and the presentation tier by one or more "client" machines. However, there is no

absolute requirement for this physical separation, providing that the logical division of responsibility outlined here is adhered to:

- **Presentation ("UI") tier**—Responsible for presenting data to the user and, optionally, for providing the mechanisms to enter and manipulate data. For human users the presentation tier will normally consist of either a traditional desktop (form-based) or a Web (browser-based) application. This tier is, by definition, the user interface and is what the vast majority of users mean when they talk about "the application." The presentation tier is the originator of requests for action and is the ultimate recipient of the results of the execution of such requests.
- **Middle ("Rules") tier**—Responsible for managing the implementation of business logic and for acting as the bridge between the presentation tier and the data. Functions here may include the security management, enforcement of business rules for data entry, or retrieval and the application of processing algorithms or logic. What can never be included in this tier is anything that requires direct communication with the end user of the application. The middle tier exposes an interface to the presentation tier through which all cross-tier communications are routed.
- **Data tier**—Responsible for the storage and management of the data. The degree of complexity at this level depends upon the capabilities of the actual database being used. The data tier exposes an interface to the middle tier through which all cross-tier communications are routed.

Okay, so what is n-tier architecture then?

The development of the World Wide Web and the requirements of distributed and scalable applications added an extra dimension to the architecture implicit in the three-tier model. The revised model, which accounts for this, is what is usually referred to as "n-tier." Strictly speaking there are still only three "tiers" (data, middle, and presentation), but the responsibilities of the middle tier have been expanded to encompass the issues associated with distributed processing and scalability—which typically involves the introduction of additional hardware.

The actual mechanisms by which an extended middle tier can be implemented are not relevant to this discussion, although for an Internet application it typically includes the addition of one or more dedicated Web servers running software whose responsibility is to manage the queuing and transmission of requests and responses between clients and the actual application servers (see **Figure 2**).

Irrespective of the architecture involved, we may still find that the simple three-layer design is not sufficient to meet our business needs. Consider the scenario in which a single business has two applications that each have their own user interfaces and business rules, but that read and write to the same (corporate) data store. Applying the three-layered design to this scenario, we end up with the model illustrated in **Figure 3**.

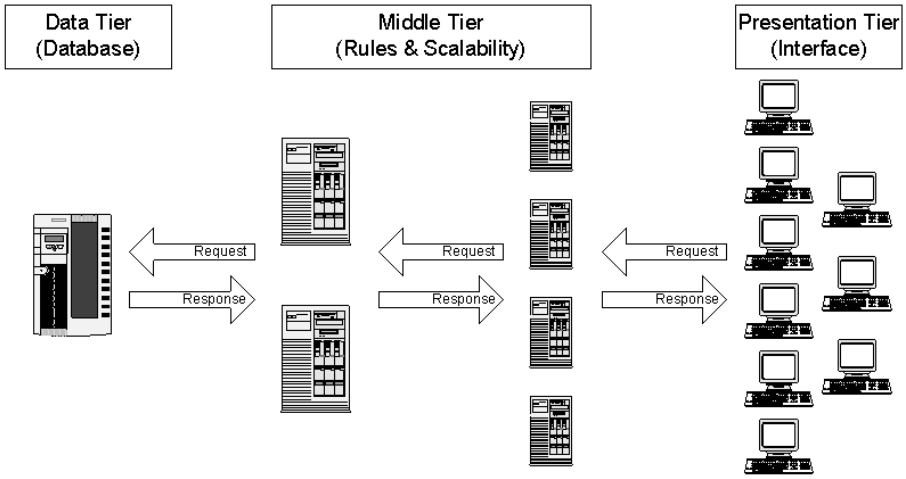


***Figure 2**. n-tier architecture.*

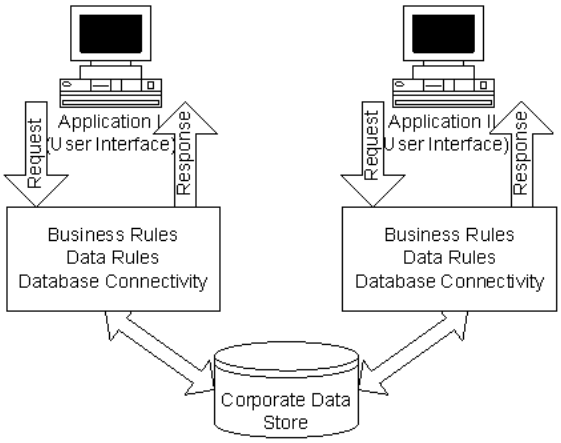


***Figure 3**. The problem with the basic three-layered design.*

As you can see, each of the application-specific rules components must also include all of the functions for handling data rules and database connectivity. But since they are both accessing the same database they are clearly both going to need the same code (after all, it is unlikely that these will vary between applications). Obviously we want to avoid duplicating the code, but we simply cannot avoid doing so as long as we restrict ourselves to only three layers.

The solution is obvious—divide the functionality into smaller units and add more layers! The result (see **Figure 4**) is a design that uses multiple layers, each of which is implemented as a discrete component that delivers a specific set of services. This allows us to design and build standard layers for such things as database connectivity and business data rules, and to share them between applications.

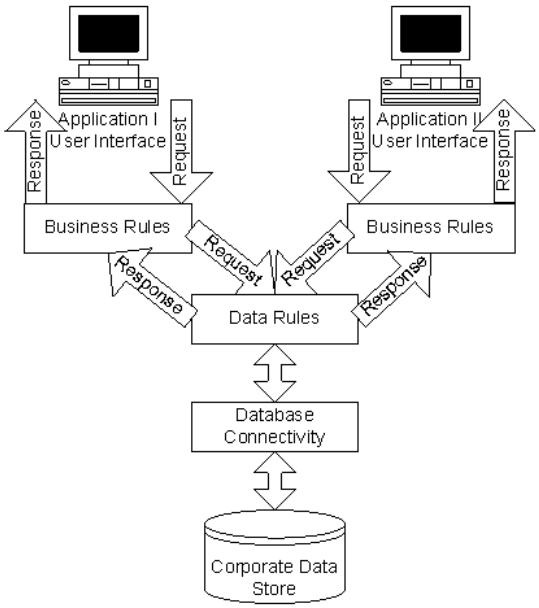


***Figure 4**. Implementing a layered design.*

What we have actually drawn here is an illustration of one of the fundamental architectural design patterns, the Layer pattern. The key element of the Layer pattern is that the flow of information is always top to bottom and consists of a "request" (or "delegation") and a "response" (or "notification"). The importance of this rule is that it means that each tier only ever needs to know the interface of the tier that sits immediately below itself, and only exposes its own interface to the tier immediately above itself. This can be expressed in the generic form illustrated in **Figure 5**.

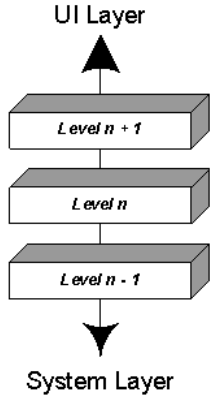


***Figure 5**. The Layer pattern.*

This leads us to the basic definition of the Layer pattern:

Layer n provides services to layer n+1 and delegates tasks to layer n-1.

There is no absolute limit to the number of layers that can be devised. As a general rule, the finer the degree of granularity that can be achieved, the easier it becomes to change functionality within a layer without affecting other parts of the system. Conversely, the greater the number of layers, the greater the total overhead incurred in packaging up and passing information between layers.

So, I should design my application using layers then?

The short answer is "yes"—in terms of application design, the Layer pattern delivers three key benefits:

- By separating business logic and data access from any specific user interface, different presentations can share common components, thereby ensuring standardization and consistency across applications.
- No layer is ever dependent upon the implementation of another. The only requirement is that each layer can communicate with the layer below. This is how we get the extensibility we were looking for.
- By separating the functionality into discrete layers, we maintain code and implement changes without having to re-write major portions of the application. By definition, components that adhere to a common interface are interchangeable.

However, like most things it is a qualified "yes" because, as **Table 1** shows, there are both pluses and minuses inherent in designing a layered system.

***Table 1**. Pluses and minuses of the Layer pattern.*

Pluses	Minuses
Functionality is separated into logical components that are coded separately.	Can be difficult to identify absolute boundaries and hence get the right degree of granularity.
Implementation is encapsulated and dependencies are localized within a layer.	Crossing layer boundaries imposes overhead and reduces efficiency.
Forces separation of high-level (for instance, UI) issues from low-level (for instance, database) issues.	Issues must be identified and assigned to the appropriate layer. Difficult to change assignments after development has started.
Increases reusability because many components can share the services of a lower level layer.	Must define basic "static" interfaces for all layers before development can start.
Team development is easier because of separation of responsibilities.	Easy to lose the sight of the "big picture." (Encapsulation = Hiding)
Separation of responsibility limits the impact of minor changes to functionality.	Changing the interface for a layer can cause changes to ripple through other layers.

Layer pattern summary

The Layer pattern provides a workable solution to the problem of designing an application for extensibility by emphasizing the necessity to insulate the implementation of the various layers

from each other. Interestingly, in this section we started with a specific instance (a monolithic application) and arrived at a generic architectural solution (the Layer pattern). In the remainder of this chapter we will show how, by identifying generic problems in software design, we can apply standard patterns to help us build specific solutions.

Implementing design patterns in Visual FoxPro

Before we dive into specific examples of how to implement specific design patterns in Visual FoxPro, we should start by defining what we mean by a "design pattern." Various definitions have been proposed and perhaps the most widely quoted is that from the seminal work *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (usually referred to as "the gang of four" or more simply as "GoF"). They offer, in the first chapter of their book, "What is a Design Pattern?", the following definition:

> *A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.*

This is a good definition because it encapsulates the four key elements of any design pattern. First, that ***it has a name***. This is vital because it allows developers to overcome one of the fundamental problems of software design—how to communicate what you are doing to others. We well remember spending nearly three-quarters of an hour sitting in a hotel lobby in Germany discussing a software problem with a colleague. It was a tortuously difficult explanation involving several sketches (yes, of course on paper napkins!), and we were struggling to understand what it was all about when suddenly something clicked. He was implementing a *Strategy* pattern! Had our friend only started out by saying that, we could have saved an awful lot of time because we would all have known immediately (at least in general terms) what the problem was, and the approach he was taking in trying to solve it.

Second, that ***it abstracts the problem***. This is referred to by "GoF" as the *intent*. It tells us both the nature of the problem and the solution described by a pattern. Consider the common problem of preventing users from creating multiple instances of an application. Typically users try to start new instances of the application because, having minimized the main screen, they forget that it is there and click the desktop icon instead of maximizing the existing instance. We can see immediately that the *Singleton* pattern is likely to be relevant because its intent is given as being to "*Ensure a class only has one instance, and provide a global point of access to it.*"

Third, that ***it defines a design structure***. It is important to realize that design patterns do not provide solutions to problems. They describe structures that allow you to solve a problem in a manner that is more likely to be reusable than if you simply wrote code to solve the problem in the context in which it arises. We have all been through the experience of realizing we have already solved a particular problem elsewhere in our code, but that we cannot reuse it because we did not isolate the solution from the situation. The purpose of a design pattern is to help you recognize situations like this and avoid them.

Fourth, that ***it identifies the distribution of responsibilities***. This is, of course, the key to all design issues and is not limited to design patterns. After all, once we know what a class (or object) has to do, writing the code is relatively easy. The benefit of a design pattern is that once we recognize a problem and match it to a pattern, the pattern tells us how we should allocate the responsibilities and therefore helps us create the "best" solution quickly.

It is not our intention, in this chapter, to offer an exhaustive review of every known design pattern (there are whole books devoted to that). We have already given an example of the *Chain of Responsibility* pattern—we used it to create generic command buttons (see Chapter 1). We have also described a data driven "factory" class for creating objects (see Chapter 2), which is actually one way of implementing the *Abstract Factory* design pattern. In the following sections we will discuss some of the most commonly encountered patterns and show how you might use Visual FoxPro to implement a solution. Let us start by looking at what has been described as the "mother of all patterns," the *Bridge*.

What is a Bridge and how do I use it?

The Bridge is the most basic of all patterns and you will find, as you become more familiar with patterns in general, that you keep finding the Bridge (in some form) at the bottom of almost every other pattern. This is why it has been referred to as the mother of all patterns.

How do I recognize where I need a Bridge?

The formal definition of the Bridge, as given by the "GoF," is:

> *Decouples an abstraction from its implementation so that the two can vary independently*

Impressive, eh? Do we really do that in our code? The short answer is that we probably *should* do so more often than we realize. Let us take a common example.

When writing code in our forms and classes we naturally want to trap for things going wrong, and usually (being considerate developers) we want to tell the user when that happens. The most obvious, and simplest, solution is to include a couple of lines of code in the appropriate method. The result might look something like this:

```
IF NOT <A Function that returns True/False>
  lcText = "Check failed to return the correct value" + CHR(13)
  lcText = lcText + "Press any key to re-enter the value"
  WAIT lcText WINDOW
  RETURN .F.
ENDIF
```

In our entire application we may well have dozens of situations where we display a wait window like this to keep the user posted as to what is going on, and this works perfectly well until one of two things happens. Either our users decide that they really hate these pesky little wait windows and would much prefer a windows-style message box, or, worse, we need to deploy the code in an environment that simply doesn't support the wait window—maybe as a COM component, or in a Web form. Now we must go and hunt through our application and find every occurrence of this code and change it to support the new requirement. We don't know about you, but in our experience, the chances of getting such a task right the first time

(not missing any occurrences and re-coding every one perfectly) are so close to zero as to be indistinguishable from it. Even if we could be confident of doing it, we still have the whole issue of testing it to deal with.

So what does this have to do with the Bridge pattern? Well, the reason that we have this problem is because we failed to recognize that we were coupling an abstraction (displaying a message to the user) to its implementation (the wait window). Had we done so we might have used a Bridge instead and then we could have avoided the problem. Here's how the same code would look if we had implemented it using a Bridge pattern:

```
IF NOT <A Function that returns True/False>
  lcText = "Check failed to return the correct value" + CHR(13)
  This.oMsgHandler.ShowMessage( lcText )
  RETURN .F.
ENDIF
```

See the difference? We no longer know, or care, how the message is going to be displayed (so we don't even need the "Press any key" line; that can be added in the message handler if it is required). All that we need to know is where to get a reference to the object that is going to handle the display for us (of course, it is a requirement that all possible handlers will implement the appropriate interface, in this case a *ShowMessage()* method). The source of this reference is the Bridge.

In this example, the *oMsgHandler* property provides the bridge between the code that requires a message and the mechanism for dealing with a message. Now, all that is needed to change the way in which our message is handled is to change the object reference stored in that property. That is something that could even be done at run time depending on the environment in which the parent object has been instantiated. This approach successfully de-couples the abstraction from its implementation, and our code is much more reusable as a result.

What are the components of a Bridge?

A Bridge has two essential components: the *abstraction*, which is the object responsible for initiating an operation, and the *implementation*, which is the object that carries it out (see **Figure 6**). The abstraction knows its implementation either because it holds a reference to it, or because it owns (that is, contains) it. Note that although the structure diagram implies that the abstraction creates the implementation, it is not an absolute requirement of the pattern. A Bridge could also make use of a pre-existing reference to an implementation object. In fact, designing Bridges this way can be very efficient because different abstraction objects can utilize the same implementation object.

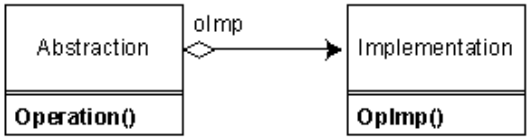


***Figure 6**. The basic Bridge pattern.*

Since Visual FoxPro has a very good containership model, most developers find that they have used it (unintentionally) to implement Bridges more often than they realize. However, don't confuse *message forwarding* with a Bridge. If a method of a command button calls a method on its parent form that directly implements the necessary action, that is message forwarding, not a Bridge. However, if that form method were to call another object to carry out the action, then we would have a Bridge.

Interestingly, another possibility occurs when a form (or other container) is maintaining a property that holds a reference to an implementation object. A contained object (for example, a command button on a form) may access that property directly and get a local reference to the implementation object. It can then call methods on the implementation object directly. Now, is this a Bridge, or message forwarding? In fact, you could probably argue either side of the case with equal justification. However, the answer doesn't really matter. This issue only arises because Visual FoxPro exposes properties as "Public" by default, and also implements back pointers that allow objects to address their parent directly. In most object-oriented environments it is simply not possible for objects to behave so rudely.

So to implement a Bridge you need to identify which object is going to play the role of the abstraction, and which the implementation. Once you have them defined, all that is left is to decide on how the Bridge between the two is going to be coded, and that will depend entirely on the scenario.

How do I implement a Bridge? (*Example: frmBridge.scx, CH15.vcx::cntTaxRoot*)

The example form (see **Figure 7**) illustrates a typical Visual FoxPro "containership bridge" in action. In this case, the form is the abstraction and is responsible for handling the display of a calculated value. A custom class, named *cntRoot*, is the implementation that is responsible for calculating the value. The form has an instance of the class (which is a simple container that has a protected *nTaxRate* property and methods named *SetRate()*, *GetRate()*, and *CalcTax()*). A custom *DoCalc()* method on the form implements the Bridge to the calculation object to get the calculated value and then updates the display accordingly. The actual code is trivial:

```
WITH ThisForm
  *** Get the base price
  lnPrice = .txtPrice.Value
  *** Here is the Bridge! Calculate the tax
  lnTax = .oCalc.CalcTax( lnPrice )
  *** And the total
  lnTotal = lnPrice + lnTax
  *** Update the display
  .txtTax.Value = lnTax
  .txtTotal.Value = lnTotal
ENDWITH
```

As you can see, the objects on the form, and the form itself, are totally insulated from the details of the process for calculating the tax rate.

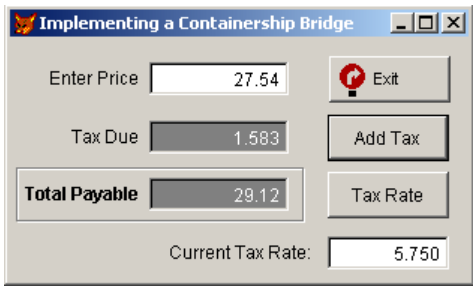


***Figure 7**. The Bridge pattern in use (FrmBridge.scx).*

All the form needs to know is that if it calls the *CalcTax()* method, it will get back a value that it can then use. By the same token, the *oCalc* object has no knowledge of what is at the other end of the Bridge and why it wants the information. It merely responds to the request for information.

A similar process is used by the form's custom *DoRate()* method, which calls on either the *oCalc* object's *GetRate()* or *SetRate()* method depending upon whether a value is entered in the "Current Rate" text box or not. Again, the form-level code is trivial:

```
WITH ThisForm
  lnRate = .txtRate.Value
  IF EMPTY( lnRate )
    *** Get the rate
    lnRate = .oCalc.GetRate()
  ELSE
    *** Set the rate
    .oCalc.SetRate( lnRate )
  ENDIF
  *** Update the display
  .txtRate.Value = lnRate
ENDWITH
```

However, the importance of this example has nothing to do with the code. What matters is that we have de-coupled the mechanism for displaying tax information from the process that calculates it—and that is the essence the Bridge pattern. For example, we can just as easily use the tax calculator from the command line, like this:

```
oCalc = NEWOBJECT( 'cntTaxRoot', 'ch15.vcx' )
oCalc.SetRate( 5.75 )
? oCalc.CalcTax( 27.54 )
```

And we could equally well change the object that the form uses, providing only that any object used by the form conforms to the expected interface (which means that it exposes methods named *CalcTax()* and *GetRate()* that return numeric values, and a method named *SetRate()* that accepts a numeric value). The two elements in this example are really independent of each other.

Bridge pattern summary

We have illustrated this Bridge by using an object dropped onto a form and named explicitly at design time. However, do remember that we could equally well have defined a property to store the name of the object, or even created the object at run time and stored the reference to it. The various mechanisms are merely implementation details; the pattern *remains the same in every situation* and that is what matters.

What is a Strategy and how do I use it?

In our introduction to this section we mentioned the Strategy pattern, but did not define it any further. The Strategy describes one solution to the main problem inherent in writing generic code—how to deal with unforeseen demands for changes in implementation.

How do I recognize where I need a Strategy?

The formal definition of the Strategy, as given by the "GoF," is:

> *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*

This is a little obscure on first reading so let us look at a common example where a Strategy pattern might help.

In the previous example we showed how to use a Bridge to de-couple the process of calculating the tax on an amount from the task of displaying it. This solution works well when there is only one possible implementation but cannot cover situations where different implementations are required at different times.

For example, the sales tax on clothing in our local area is 5.75%, but if we travel 20 miles south the tax base is only 5.25%, while if we travel into the next state (only 50 miles away) there is no tax on clothing at all. The same item of clothing ticketed at $29.95 could therefore cost us either $31.67, $31.52, or $29.95 depending on where we buy it. Stating this in more general terms, the tax applicable to the transaction depends upon the *context* (the specific combination of type of item and the location) in which the transaction takes place. If we were writing code into an application to handle this, we might start out with something like this:

```
DO CASE
  CASE lclocale = "Akron"
    IF lcItemType = "Clothing"
      lnTaxRate = 5.75
    ELSE
      *** Other items here
    ENDIF
  CASE lclocale = "Canton"
    IF lcItemType = "Clothing"
      lnTaxRate = 5.25
    ELSE
      *** Other items here
    ENDIF
```

```
  CASE lclocale = "Grove City"
    IF lcItemType = "Clothing"
      lnTaxRate = 0.00
    ELSE
      *** Other items here
    ENDIF
  OTHERWISE
    *** Apply a Default value
    lnTaxRate = 5.50
ENDCASE
```

We can see immediately what one problem is going to be! What happens when we need to add "Cleveland" to our list of locales, or when Akron, dismayed at the loss of clothing sales to Canton, cuts its tax rate for clothing? We need to change our code with all the concomitant risks involved. Moreover, as we increase the number of locations, this code will rapidly become unmanageable. Of course, we could store this information in a table (one column for location, and one column for each item type perhaps) and look it up each time that we needed it. That way we would only have to modify records when data changed, but it could quickly become a very large table indeed and there would be a lot of duplication and redundancy in the data, so even this is not an ideal solution.

What we really want our application to do is to be able to pass the ticket price, and context information, to something that will simply tell us what the tax should be. However, unless we simply move the code (which re-locates, but does not change the problem), a Bridge is not the solution because it only has a single implementation.

The Strategy pattern allows us to define classes for each situation that we must deal with and then instantiate the appropriate one at run time. That would mean that we could replace all of the preceding code in our application with just one line that never needed to change no matter what the situation:

```
lnTax = This.oCalcTax( nPrice, lcLocale , lcItemType )
```

What are the components of a Strategy?

A Strategy has three essential components: the "abstract strategy," which is the class that defines the interface and generic functionality for the "concrete strategies," which are the subclasses that define the various possible implementations. The third component, the "context," is responsible for managing the reference to the current implementation (see **Figure 8**).

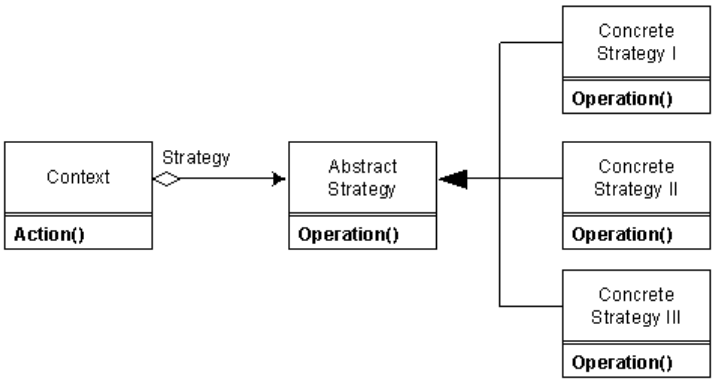


***Figure 8**. The basic Strategy pattern.*

If you are thinking that this diagram looks very similar to the Bridge, you are correct. You can even think of a strategy as a "dynamic bridge" in which one end (the *context*) is static, but the other (the *strategy*) is being created from among a number possible of variants in order to deliver a specific implementation at a given moment. The essence of the pattern is that the decision as to which subclass should be instantiated at any time depends upon the state of the *client*. The classic example of the pattern lays the responsibility for instantiating the concrete strategy object on the client, which then passes a reference to the context.

In order to implement a Strategy pattern you need to define the abstract strategy class and as many different specific subclasses as you need. The object, which is going to play the role of the context (in VFP this is typically the form, or parent container), needs to expose an appropriate interface to its potential clients and also requires a property that is used to store the reference to the currently active implementation object.

How do I implement a Strategy? (*Example: frmStrat.scx, CH15.vcx::cntTaxStrat01...04*)
The example form (see **Figure 9**) illustrates how we might use a Strategy pattern to implement a solution to the sales tax issue that we discussed in the preamble to this topic. In this case the Add Tax button is the client and the form is the context.

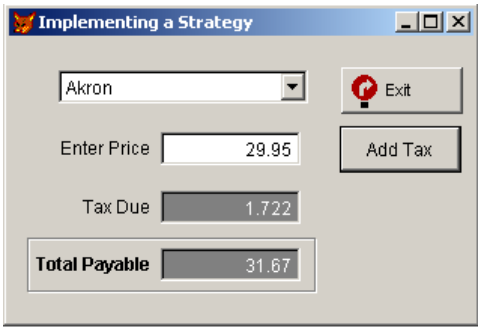


***Figure 9**. The Strategy pattern in use (FrmStrat.scx).*

The *cntTaxRoot* class that we used for the Bridge example in the preceding section has been subclassed four times (classes *cntTaxStrat01* through *cntTaxStrat04*), and each subclass has been pre-defined with a specific tax rate (by setting the *nTaxRate* property in the subclass). Finally, the form's *Load()* method is used to create and populate a simple cursor with a list of cities that are keyed to one of the four *cntTaxStrat* subclasses—this cursor is used as the source for the dropdown list.

When the Add Tax button is clicked, the following code, in its *OnClick()* method, is executed. This simply gets the location ID and the current price from the form controls and passes them to the form's custom *DoCalc()* method. The form's *DoCalc()* method returns the appropriate amount of tax, and the remainder of the code updates the display accordingly.

```
LOCAL lnPrice, lnTax, lnTotal
STORE 0 TO lnPrice, lnTax, lnTotal
WITH ThisForm
  *** Get "context" (Location and Price)
  lcLocn  = ALLTRIM( .cbolocation.value )
  lnPrice = .txtPrice.Value

  *** Get the tax using the strategy
  lnTax = .DoCalc( lcLocn , lnPrice )

  *** Calculate the total
  lnTotal = lnPrice + lnTax

  *** Update the display
  .txtTax.Value = lnTax
  .txtTotal.Value = lnTotal
ENDWITH
```

All the work is obviously being done in the form's *DoCalc()* method. However, there is not as much code there as you might expect. It expects to receive the two parameters—the context key (which, since it is coming from a dropdown list, will be a character string) and the price on which the tax is to be calculated.

The context key is used to generate the name of the required subclass. If the currently instantiated class is not right, we simply instantiate the correct one. Naturally, all of the subclasses inherit the *CalcTax()* method. That, you will recall, uses whatever is set in its *nTaxRate* property if no rate is passed explicitly. So, we can get the correct amount of tax by simply calling the subclass's *CalcTax()* method with just the price we were given. The return value is then returned to the client.

```
LPARAMETERS tcContext, tnPrice
WITH ThisForm
  *** Define the name of the strategy object
  lcStrategy = "CNTTAXSTRAT" + PADL( tcContext, 2, '0' )
  *** Is it already there
  IF ISNULL( .oStrategy ) OR NOT UPPER( .oStrategy.Name ) == lcStrategy
    *** We need to create this one
    .oStrategy = NEWOBJECT( lcStrategy, 'Ch15.vcx' )
  ENDIF
  *** Now just call it's CalcTax() method and pass the price
  lnTax = .oStrategy.CalcTax( tnPrice )
  RETURN lnTax
ENDWITH
```

The example illustrates one way of implementing a strategy—in which we have made the context object responsible for both the instantiation and the maintenance of the strategy object. That makes sense in the scenario and with the interface illustrated. Other scenarios may require different implementations.

For example, consider the application of discounts to an order. The order entry form (the context) may need to apply several different kinds of discount to an order:

- Item quantity discounts, applied to a single line item at a time
- Order value discount, applied to the total order value
- Special item discounts or promotions
- Customer discount applied to order value based on the customer

One possible interface would use command buttons (the clients) so that the operator can determine when to apply a specific type of discount. In that scenario it would be entirely appropriate, and much simpler, for each button to be responsible for instantiating the appropriate discount strategy subclass and storing its reference to the form property. (Remember that while we can do this sort of thing easily in Visual FoxPro, other languages would have to pass the reference explicitly.) The form could then handle the actual calculation and deal with displaying the result so that the code does not have to be duplicated in every button.

Strategy pattern summary

We have shown one possible implementation of the Strategy pattern and outlined another. The benefit of the Strategy is that it avoids the necessity of having to embed alternative implementations in code, allowing us to create separate objects, which share a common interface, for each option and to only instantiate the necessary object at run time. As with other patterns, the actual implementation details may vary with circumstances but the pattern itself does not change.

What is a Chain of Responsibility and how do I use it?

In the preceding section we stated that a Strategy pattern described one solution to the problem of dealing with alternative implementations at run time. The Chain of Responsibility is another way to tackle the same basic problem.

How do I recognize where I need a Chain of Responsibility?

The formal definition of the chain of responsibility, as given by the "GoF," is:

> *Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.*

In the previous example we showed how to use a Strategy to cope with the problem of applying a location-specific rate of sales tax at run time. However, as we have seen, in order to

implement a Strategy some object, somewhere, has to decide, at run time, which of the possible subclasses to implement. This may not always be either desirable, or even possible.

In a Chain of Responsibility, each object knows how to evaluate a request for action and if it cannot handle the request itself, knows only how to pass it on to another object, hence the "chain." The consequence is that the client (which initiates the request for action) now only needs to know about the very first object in the chain. Moreover, each object in the chain only needs to know about one object too—the next one in the chain. The Chain of Responsibility can either be implemented using a predefined, or *static*, chain or the chain can be built at run time by having each object add its own successor when necessary.

We have already met one example of a static Chain of Responsibility in Chapter 1, where we used it to create generic command buttons. In that case the chain was pre-defined because the button first looked for a specific method on its immediate parent and, if it was found, passed the required action to it. Otherwise, it routed the request directly to the form, which, as part of its interface, provides a "default" behavior (in that case, it was to display an "under construction" message). Another implementation is described by Doug Hennig in his paper "Error Handling in Visual FoxPro" (you can download this excellent paper from the Technical Papers section at: www.stonefield.com).

What are the components of a Chain of Responsibility?

A Chain of Responsibility can be implemented by creating an abstract "handler" class (to specify the interface and generic functionality) and creating concrete subclasses to define the various possible implementations (see **Figure 10**). However, there is no absolute requirement for all members of a Chain of Responsibility to be descended from the same class, providing that they all support the necessary interfaces to integrate with other members of the chain. Client objects need a reference to the specific subclass that is their individual entry point into the chain. (Note that not all clients need use the same entry point!!)

Notice that, yet again, we see the basic Bridge pattern here, because each link in the chain is actually a bridge between an abstraction and an implementation. All that is different is that a single object can play both roles depending on its situation. Thus the first link has the client as the abstraction, and the first concrete handler as the implementation. However, the second link now has the first handler playing the role of the abstraction, and the second handler the implementation. This pattern can, in theory, at least, be repeated *ad infinitum*.

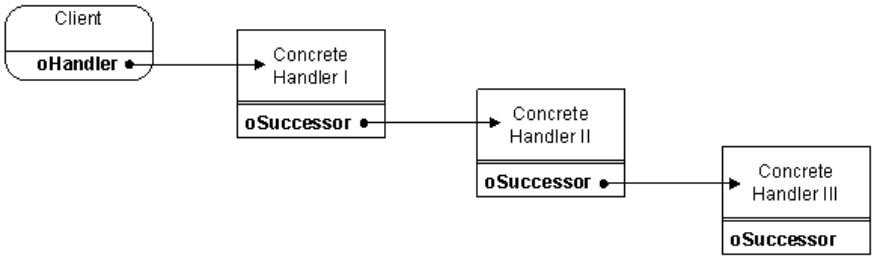


***Figure 10**. The basic Chain of Responsibility pattern.*

In order to implement a Chain of Responsibility pattern, you need to define an abstract handler class and as many different specific subclasses as you need. The difference from the

Strategy pattern is that the abstract class must now define the mechanism by which an object can determine whether it should handle the request or pass it on, and a property to hold a reference to the next object in the chain. In fact, there is no absolute requirement for all handlers to inherit from the same abstract handler, providing that they adhere to the minimum defined interface.

How do I implement a Chain of Responsibility? (*Example: frmChor.scx, CH15.vcx::cntTaxChain01...04*)

The example form (see **Figure 11**) illustrates how we might use a dynamic Chain of Responsibility pattern to implement a solution to the sales tax issue that we discussed in the previous section. In this case the Add Tax button delegates responsibility to the form by calling its custom *DoCalc()* method. The form acts as the client and holds a reference to the first object in the chain, which is created, when needed, explicitly in the *DoCalc()* method.

```
LPARAMETERS tcContext, tnPrice
LOCAL lnTax
WITH ThisForm
  *** Check that we have the first object in the chain available
  IF VARTYPE( This.oCalc ) # "O"
    *** We don't so create it
    .oCalc = NEWOBJECT( 'cntTaxChain01', 'ch15.vcx' )
  ENDIF
  *** Now just call ProcessRequest() method and pass both context and price
  lnTax = .oCalc.ProcessRequest( tcContext, tnPrice )
  IF ISNULL( lnTax )
    *** Couldn't process the Request at all
    MESSAGEBOX( 'Unable to Process this Location', 16, 'Failed' )
    lnTax = 0
  ENDIF
  RETURN lnTax
ENDWITH
```

Notice that the code in the Add Tax button on the form is identical to that required for the strategy (FRMSTRAT.SCX), but the code in this form's *DoCalc()* method is slightly different.

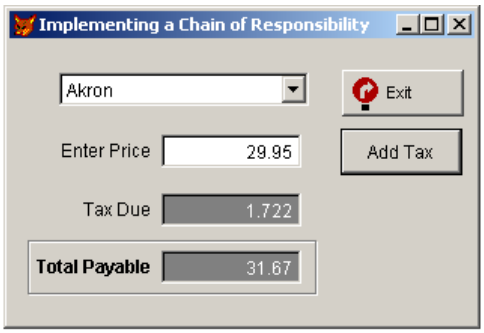


***Figure 11**. The Chain of Responsibility in use (FrmChor.scx).*

The difference is in the classes used to carry out the tax calculation. For this example we have created a new subclass of the *cntTaxRoot* class, named *cntTaxChain*. This class adds four protected properties (see **Table 2**) and one exposed method named *ProcessRequest*.

***Table 2**. Properties for the Chain of Responsibility handler class.*

Property	Description
cCanHandle	Property used to define the context handled by this subclass
cNextObj	Name of the class to instantiate as next object in the chain
cNextObjLib	Class library for next object in the chain
oNext	Object reference to the next object in the chain

The *ProcessRequest()* method is responsible for determining whether the incoming request is to be handled locally. If so, it simply calls the default *CalcTax()* method defined in the original root class. If not, the action taken depends upon whether another object is defined, and available, to handle the request, as follows:

```
LPARAMETERS tcContext, tnPrice
LOCAL lnTax
WITH This
  *** Can we deal with the request here?
  lcCanHandle = CHRTRAN( .cCanHandle, "'", "" )
  IF tcContext = lcCanHandle
    *** Yes, so call standard CalcTax() method, passing Price
    lnTax = .CalcTax( tnPrice )
  ELSE
    *** We cannot deal with it, Do we have an object defined to pass it on to?
    IF NOT EMPTY( .cNextObj ) AND NOT EMPTY( .cNextObjLib )
      *** Yes we do. but does it already exist
      IF VARTYPE( This.oNext ) # "O"
        *** Create the object and call it
        .oNext = NEWOBJECT( .cNextObj, .cNextObjLib )
      ENDIF
      *** And just call on the specified object
      lnTax = This.oNext.ProcessRequest( tcContext, tnPrice )
    ELSE
      *** Nowhere else to go, just Return NULL
      lnTax = NULL
    ENDIF
  ENDIF
  RETURN lnTax
ENDWITH
```

This is all the code that is needed. The individual subclasses for this very simple example have no custom code at all; everything is handled by setting properties (including the *nTaxRate* property defined in the original root class). Note that if you run the example as provided, and select "Cleveland" as the location, you will get the failure message defined by the form's *DoCalc()* method. This is because to process the Cleveland location we would need the class named *cntTaxChain04*, which is not named anywhere in the chain. The chain ends because the *cntTaxChain03* class has not had its *cNextObj* and *cNextObjLib* properties defined. To remedy this, simply set these two properties to point to *cntTaxChain04* and *CH15.VCX*, respectively. "Cleveland" then becomes available as a valid location.

Although we have defined the chain sequentially, there is no real reason to do so, nor is there any absolute requirement to use properties to define the next object the way in which we have done here. Another possible (and very flexible) implementation would be to use a data table to store details of handler classes and simply have each handler object look up a keyword to find the details of the next object to instantiate.

Chain of Responsibility pattern summary

The Chain of Responsibility provides us with another way of resolving the problem of providing functionality without the necessity to code it explicitly. Perhaps the biggest benefit of the Chain of Responsibility is the ease with which it can be extended, while its main drawback is that it can dramatically increase the number of objects active in the system. As always, the final word is to remind you that even though the actual implementation details may vary, the pattern itself does not change.

What is a Mediator and how do I use it?

The Mediator describes one solution to a key issue that we all encounter whenever we try to design a new class: how to deal with situations where one object has to respond to changes in, or control the behavior of, another.

How do I recognize where I need a Mediator?

The formal definition of the Mediator, as given by the "GoF," is:

> *Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.*

This addresses one of the most fundamental problems that we have to tackle in any software development task—that of ensuring that objects can communicate with each other without actually having to include hard-coded references in our classes.

Nowhere is this more critical than when building the complex user interfaces that the current generation of computer-literate end users not only expect, but demand. Typically we have to make the entire UI respond as a single entity, enabling and disabling functions and controls in response to the user's actions and choices, or according to their rights and permissions. At the same time we want to design and build generic, reusable classes. The two requirements are, apparently, in direct conflict with one another.

Take a look at the example form that we used to illustrate the Chain of Responsibility in the preceding section (Figure 11). If you run this example, you will notice that the Add Tax button is enabled when the form is instantiated even though there is no price specified. Of course, clicking the button with a price of $0.00 simply returns $0 and apparently nothing happens anyway. But what happens if we enter a negative value? The short answer is that everything just works, so that if you enter $-29.95 as the price and click the Add Tax button, the form tells you that the tax due in Akron is $-1.722 and the total price is $-31.67.

It might be better if the Add Tax button were only enabled when a price that was greater than 0 had been specified. Of course, the simplest solution is just to add a couple of lines of code to the *Valid()* of the textbox in this form that would implement functionality like this:

```
IF This.Value > 0
  ThisForm.cmdCalc.Enabled = .T.
ELSE
  ThisForm.cmdCalc.Enabled = .F.
ENDIF
```

This sort of "tight coupling" may be acceptable when we are dealing with one textbox and one command button in a single form. However, we will quickly run into trouble if we try to adopt this solution when dealing with multiple controls that have to interact in complex and differing combinations. Even finding where the code that controls a particular interaction is specified can be a problem, and simply changing the name of one object becomes a major undertaking. This is where the Mediator pattern comes into its own.

The basic idea is that each object communicates with a central "mediator," which knows about all of the objects that are currently in scope and how to manage their state when a given event is reported. In this way we avoid all of the issues associated with placing specific code into the method associated with the *Valid()* event of a control. Instead we can write completely generic code in its parent class. So, if we were using a mediator object, we could replace the specific code in the instance of the price textbox with code like this in its parent class:

```
This.oMediator.StateChange( This )
```

As you can see, the textbox has no idea what the mediator will do with the information, or even what information it wants. All that it has to do is to call the *StateChange* method and pass a reference to itself. Any subsequent action is up to the specific implementation of the Mediator.

What are the components of a Mediator?

A Mediator has two essential requirements. First, we need a *mediator* class that defines the interface, and generic functionality, for the *concrete mediators* that are the subclasses that define the various possible implementations. The second is that all Colleague objects must be based on classes that can communicate with their Mediator. The basic structure for the Mediator pattern is shown in **Figure 12**.

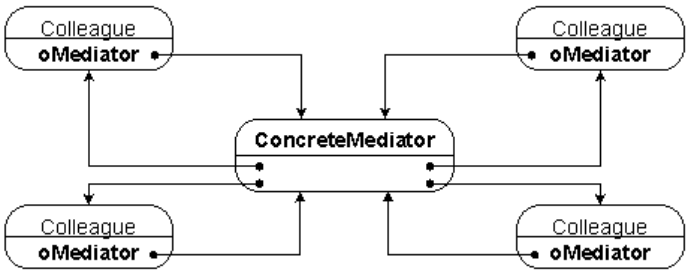


***Figure 12**. The basic Mediator pattern.*

You can see from the diagram that classes that need to work with a mediator need to hold a reference to the mediator object. You are probably thinking that in Visual FoxPro we have a ready-made candidate for the mediator class in the Form (in the context of the UI anyway).

Using the form as the mediator is nice because it is always available to any object through the *ThisForm* reference without the need to worry about specific properties. However, when dealing with non-visual classes this may not be the best solution, so it is probably better to define a generic, non-visual, mediator class.

The second thing that you will notice from the diagram is that the mediator object needs to hold a reference to all of its colleagues. This raises the question of how it should acquire and hold those references—to which there are several possible answers. Perhaps the simplest to implement is that each control class defines a *RegisterMe()* method that, whenever an instance of the class is created, checks for the presence of a mediator and passes a reference to itself to the mediator. Another possibility is that the mediator class defines a *GoFindEm()* method that searches its environment to discover objects. Either way, the mediator class will also need to maintain a collection (array) to store references to its colleagues.

How do I implement a Mediator? (*Example: frmMed.scx, CH15.vcx*)

In order to implement a Mediator for our simple little tax example, we have quite a lot of preparation to do. First we will need a form class that can handle a mediator. The class *frmMediated* in CH15.VCX has had three custom properties added (see **Table 3**).

***Table 3**. Properties for the Mediated Form class.*

Property	Description
cmedclass	Name of the mediator class to instantiate for this form
cmedlib	Class library for the mediator class to be instantiated for this form
omediator	Exposed property to hold the reference to the mediator object

It has the following code added to its *Load()* event to ensure that the defined Mediator is in place before any other form control gets instantiated.

```
LOCAL lcMClass, lcMLib
DODEFAULT()
WITH ThisForm
  *** If a mediator class is specified, instantiate it
  lcMClass = ALLTRIM( .cMedClass )
  lcMLib = ALLTRIM( .cMedLib )
  IF NOT EMPTY( lcMClass ) AND NOT EMPTY( lcMLib )
    .oMediator = NEWOBJECT( lcMClass, lcMLib )
  ENDIF
ENDWITH
```

Of course, we also have to be careful about cleaning up object references, so the form class includes the following code in its *Destroy()* event, to make sure that the clean up happens

```
IF VARTYPE( This.oMediator ) = "O"
  This.oMediator.Destroy()
ENDIF
```

This is required to avoid the problem that arises because the normal sequence of events is that objects are destroyed in the reverse order of their creation. Since the mediator is created first it would normally be destroyed last, but because it holds references to other objects they cannot be destroyed while it exists. So we need to force the mediator's destruction "out of turn" when the form is being released so that the objects can release themselves properly.

Next, we will need to define new subclasses for each of the controls that will have to work with the mediator. A new custom property (*oMediator*) has been defined to hold a local reference to the mediator object, together with three new methods:

- *Register ()* Called from the *Init()* of the control. Checks for the presence of a mediator. If one is found it stores a local reference to it and then calls mediator's *Register()* method, passing a reference to itself.
- *Notify()* Method that calls the *Notify()* method on the mediator and passes a reference to the current control.
- *UnRegister()* Called from the *Destroy()* of the control. Checks the local reference to the mediator. If found, calls the mediator's *UnRegister()* method, passing a reference to itself and then releases the local reference.

In addition, the following events (where applicable) have been modified to call the control's *Notify()* method: either *InterActiveChange()* and *ProgrammaticChange()* for lists or *Valid()* for textboxes.

Then we need an abstract mediator class that defines the interface and generic behavior for the *Register()* method and an *aObjects* collection. This class also defines *StateChange()* as a template method, to be implemented in the subclass. We can then create a specific subclass of this to implement the specific behavior that we want in our form (which was, you may recall, to only enable the Add Tax button when the Price textbox has a value that is greater than 0).

Finally, we have to re-create the form using the new classes.

Phew! Seems like a lot of work ,doesn't it? However, you will notice that everything except the creation of the concrete mediator that implements the specific behavior is entirely generic and is, therefore, completely reusable. In other words, it only has to be done once.

It is important to emphasize just how flexible we have now made this simple little form. All of the code that governs the interaction between its controls is confined in a single subclass that is specific to the form, but can be maintained outside of it. To change the behavior we can either modify the existing subclass, or create a new one and implement it by just changing a couple of properties on the form. **Figure 13** shows the form just after instantiation now that it implements the mediator—notice that the Add Tax button is disabled.

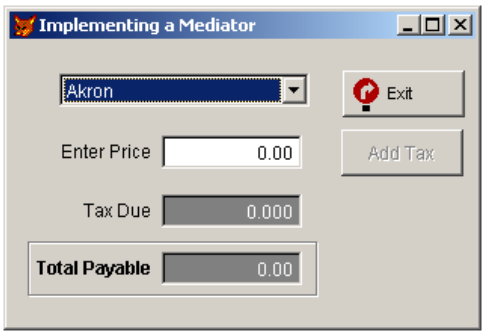


***Figure 13**. The Mediator in use (FrmMed.scx).*

The only instance level code we need in the form is in the *Init()* event because we want to force the controls to initialize themselves correctly. This cannot be done before the Form's *Init()* because that is the first moment at which we can be sure that all the controls on the form have been instantiated and have been registered with the mediator.

The abstract mediator class is defined in CH15.VCX as *xMediator* along with a concrete subclass (named *medTaxForm*) that is used in the example form. The code in the abstract mediator is pretty straightforward and is concerned with building and managing the collection of registered objects. The only thing to note here is that we are using a three-column collection (name, parent, and object reference) to allow for the fact that a complex form might have more than one object with the same name, but in different containers.

The code in the *medTaxForm.Notify()* method is, of course, specific to this particular form, as shown next. The first thing that it does is to determine which object has called it. If it is the combo box (which is also mediator-enabled), nothing happens. However, if the call is from the Price textbox, then the code retrieves, from the mediator's collection, the object reference to the Add Tax button, which in this form is actually named *cmdCalc*.

```
LPARAMETERS toObjRef
LOCAL lcName
*** Get the name into a local Variable
lcName = UPPER( ALLTRIM( toObjRef.Name ))
*** If this is a message from the Price TextBox
IF lcName = 'TXTPRICE'
  *** We need a reference to the 'Add Tax' command button
  lnRow = This.FindInCollection( 'CMDCALC' )
  IF lnRow > 0
    *** Got it! Grab a reference to it
    loTarget = This.aRegistered[ lnRow, 3 ]
    *** Enable the button according to the value
    loTarget.cObjmode= IIF( toObjRef.Value > 0, "E", "D" )
    loTarget.Refresh()
  ENDIF
ENDIF
*** Just return from here
RETURN
```

The object reference is used to set the *mode* property of the button according to the current value in the price textbox, and to call its *Refresh()* method to implement the change.

That is all that is needed in this (admittedly simple) example. However, once the necessary classes have been established, even very complex interactions can be managed by using the *Notify()* method as a control method and adding additional methods to the subclass as needed.

Mediator pattern summary

Implementing the Mediator pattern undoubtedly requires more planning, and much more preparation, than any of the patterns we have discussed so far. However, in situations where you have to manage complex interactions, it amply repays the effort in three ways. First, it minimizes subclassing by localizing behavior that would otherwise be spread among several classes in a single object. Second, it avoids the necessity for tight coupling between objects. Third, it simplifies the logic by replacing *many-to-many* interactions between individual controls with *one-to-many* interactions between the mediator and colleagues. The main drawback to the pattern is that, especially when dealing with large numbers of interactions, the mediator itself can get very complicated and difficult to maintain.

What is a Decorator and how do I use it?

The Decorator describes a solution to the problem of adding functionality to an object without actually changing any of the code in that object.

How do I recognize where I need a Decorator?

The formal definition of the decorator, as given by the "GoF," is:

Attach additional responsibilities to an object dynamically

The need to alter the functionality or behavior of an object dynamically typically arises in either of two situations. First, where the source code for the object is simply not available; perhaps the object is an ActiveX control or a third-party class library is being used. Second, where the class is widely used but the specific responsibility is only needed in one (or more) particular situations and simply to add the code to the class would not be appropriate.

In the preceding sections we have looked at various ways of determining the amount of tax to apply to a *price* depending on the *location*. Our basic tax calculation object (*CH15.vcx:: cntTaxRoot*) works by exposing a method named *CalcTax()*, which takes two parameters, a value and a rate, and returns the tax due. We tackled the basic problem of dealing with tax in a form by using the Bridge pattern to separate the implementation from the interface. However, as we quickly saw, that was not flexible enough to cope with different tax rates in different locations. Both a Strategy and a Chain of Responsibility can handle the issue; however, both solutions involve subclassing our basic tax calculator class.

The Decorator pattern allows us to solve the same problem without the need to subclass the original. Instead, we define a new object that has exactly the same interface as the tax calculator, but that includes the necessary code to determine the appropriate tax rate for a given location. This object "looks" just like the tax calculator to its client, but, because it also holds a reference to the real tax calculator, it can "pre-process" any request for a tax rate, and then simply call the real implementation when ready.

What are the components of a Decorator?

A Decorator has two essential requirements. First, we need the implementation class that defines the interface, and the core functionality. Second, we need a Decorator that reproduces the interface of the implementation, and holds a reference to it. The client object now directs calls that would have gone directly to the implementation to the decorator object instead. The basic structure for the Decorator pattern is shown in **Figure 14**.

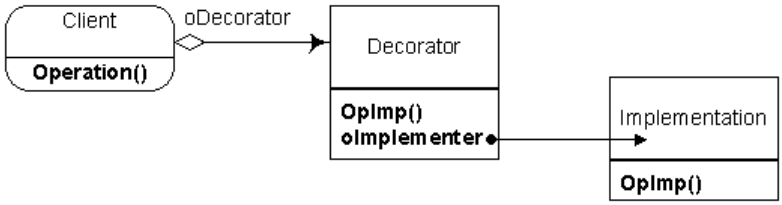


***Figure 14**. The basic Decorator pattern.*

This pattern is actually an extended bridge. As far as the client is concerned it can address the decorator object as if it really were the implementation at the end of a standard bridge, because the interface of the two is the same. As far as the implementation is concerned, the request looks exactly the same as if it had come from directly from the client. It does not ever need to know that the decorator even exists.

How do I implement a Decorator?

(*Example: frmDecorator.scx, CH15.vcx:: cntDecorator*)

A new class, named *cntDecorator*, has been defined to implement the Decorator example. It has three custom properties and one custom method as shown in **Table 4**.

***Table 4**. Custom PEMs for the decorator class.*

Name	PEM	Description
cImpclass	Property	Name of the class to instantiate for this decorator.
cImplib	Property	Class library for the implementation class to instantiate.
oCalculator	Property	Object reference to an instance of the class that is the real implementer of the *CalcTax()* method. This object is instantiated in the decorator's *Init()*.
CalcTax	Method	The decorator's implementation for the equivalent method on the "real" implementer.

The decorator class is actually very simple. When it is created, it instantiates the real implementation class, which is defined by its class name and library properties. That object must, of course, implement a *CalcTax()* method that returns a numeric value.

The code in the decorator's *CalcTax()* method is quite straightforward. It expects to receive two parameters—the location ID as a string, and the price for which the tax is required. Notice that these are not the same parameters that are required by the *CalcTax()* method in the real class. There we need to pass a price and a tax rate to apply to it. The decorator's *CalcTax()* method determines the appropriate rate based on the location ID and then calls its implementation object's *CalcTax()* method passing the *real* parameters. The return value is just passed back to the client without any further modification.

```
LPARAMETERS tcLocation, tnPrice
LOCAL lnRate, lnTax
STORE 0 TO lnRate, lnTax

*** Determine the correct rate
DO CASE
  CASE tcLocation = '01'
    lnRate = 5.75
  CASE tcLocation = '02'
    lnRate = 5.25
  CASE tcLocation = '03'
    lnRate = 0.00
  OTHERWISE
    lnRate = 0
ENDCASE

*** Now pass on the call to the real object
lnTax = This.oCalculator.CalcTax( tnPrice, lnRate )

*** And return the result
RETURN lnTax
```

The example form (see **Figure 15**) uses shows the Decorator in use.

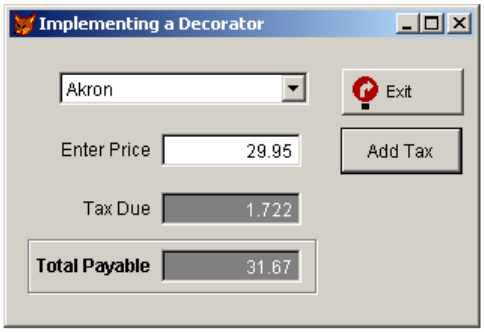


***Figure 15**. Using a decorator (frmDecorator.scx).*

The Decorator example uses a copy of the form that we created to illustrate the Mediator pattern in the previous section. The only change is to two lines of code in the Form's *DoCalc()* method where, instead of instantiating the first member of the Chain of Responsibility and calling its *ProcessRequest()* method, we now instantiate the decorator and call its *CalcTax()* method.

```
LPARAMETERS tcContext, tnPrice
LOCAL lnTax
WITH ThisForm
  *** Check that we have the Decorator available
  IF VARTYPE( This.oCalc ) # "O"
    *** We don't so create it
    .oCalc = NEWOBJECT( 'cntDecorator', 'ch15.vcx' )
  ENDIF
  *** Now just call it's CalcTax() method and pass both location and price
  lnTax = .oCalc.CalcTax( tcContext, tnPrice )
```

```
  IF ISNULL( lnTax )
    *** Couldn't process the Request at all
    MESSAGEBOX( 'Unable to Process this Location', 16, 'Failed' )
    lnTax = 0
  ENDIF
  RETURN lnTax
ENDWITH
```

Although this is a very simple example, you can see how easy it would be to extend the functionality of the Decorator. One very common "real life" problem that this pattern can be used to address is how to provide implementation specific validation to a generic save routine.

Decorator pattern summary

The Decorator pattern is used when we wish to modify the basic behavior of a specific instance without the necessity of either creating a new subclass, or changing the code in the original. Essentially the Decorator acts as a pre-processor for its implementation by interposing itself between its client and the implementation.

What is an Adapter and how do I use it?

The Adapter, as its name implies, describes a solution to the problem of making two disparate interfaces compatible by acting as a translation mechanism between them. The key distinction between an Adapter and a Decorator is that while a Decorator modifies behavior, the Adapter only modifies an interface.

How do I recognize where I need an Adapter?

The formal definition of the Adapter, as given by the "GoF," is:

Convert the interface of a class into another interface clients expect

The need for an Adapter usually results from modifications or enhancements to class libraries or COM components. For example, re-factoring can often result in significant changes to the way in which the interface for a class needs to be defined. This in turn may necessitate major changes in application code that calls on the services of such classes. Making such changes is always a high-risk option and should be avoided wherever possible.

One solution when source code is available is to retain the methods in the original interface to the class, but remove the implementation into new methods. The old methods are then left as "control" methods that no longer do the real work; instead, they manage the calls to other methods. However, this still requires that existing code be modified.

The alternative is to create a new class that replicates the expected interface and translates calls to that interface into the correct format for the replacement. When dealing with components to which the source code is not available, this is the only method of dealing with changes to their interface.

What are the components of an Adapter?

An Adapter has two essential requirements. First, we need a class that defines the interface that is to be adapted, the *adaptee*. Second, we need the *adapter* class that defines the expected interface, and maps the methods in that interface to those defined by the adaptee. The client

calls one of the "expected" methods, which is exposed on one side of the adapter, which simply passes the method call on to the appropriate method in the adaptee. The basic structure for the Adapter pattern is shown in **Figure 16**.

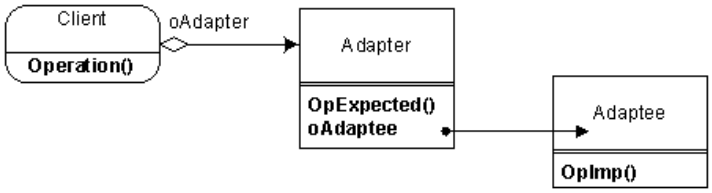


***Figure 16**. The basic Adapter pattern.*

This pattern is actually another variant of the basic bridge. As far as the client is concerned it can address the adapter object as if it really were the implementation at the end of a standard bridge. Since the adapter supports the expected interface, the client remains unaware that the implementation is actually being handled by another method as part of a different interface.

How do I implement an Adapter?

An adapter is typically implemented by creating a subclass of the adaptee and allowing it to inherit the methods of expected interface. These can then be coded with the appropriate calls to methods defined by the adaptee.

This presents somewhat of a problem in Visual FoxPro because it does not support the multiple inheritance that is required to do it this way. A new feature that was introduced in Visual FoxPro 7.0 is the `IMPLEMENTS` keyword that allows a Visual FoxPro class, defined in code using the `DEFINE CLASS` command, to inherit an interface that is defined in a type library. However, because it relies on type libraries, it can only be used with COM components, and not with native Visual FoxPro classes, so it is not really much help in this context.

The best way to implement an adapter in Visual FoxPro is to create a class that exposes the methods of the expected interface, and holds an object reference to the adaptee. Client objects can then call their expected methods that in turn call the appropriate method on the adaptee. The result is so nearly identical to the implementation of a decorator illustrated in the preceding section that we have not created a specific example for it. The only difference from the decorator is that instead of modifying the behavior, the adapter simply re-routes the method call.

Adapter pattern summary

The Adapter pattern is used to avoid the necessity of changing code when an interface is changed, or to allow for future modifications or implementations when designing generic classes. However, because Visual FoxPro does not support multiple inheritance, the only implementation mechanism available is identical to that used by the Decorator.

What is a wrapper, and how do I use it?

Now, that is a very good question indeed! If you study *Design Patterns*, you will not find a pattern named "wrapper" anywhere in the list of primary patterns. However, a more diligent

search of that resource will reveal that it is actually listed as a synonym for both the Adapter and Decorator patterns.

Now, this does not make much sense to us. If a wrapper really is a synonym for both, then logically they are also synonyms for each other and it follows that all three terms *must* refer to the same pattern. Yet this is clearly *not* the case! As we have seen, Decorator and Adapter are quite different in their intent, and any similarities in their implementation are, as always, irrelevant in the context of a pattern's definition. While not wishing to disagree with the "GoF," we feel that, in this particular case at least, they have it wrong.

It seems to us that since the word "wrapper" actually means a covering, to describe either the Decorator or the Adapter as a "wrapper" is also linguistically inaccurate. Both of these patterns rely on "fooling" the client into believing that an interposed object is actually the intended target by having it look just like the target. They may, therefore, both be *mimics*, or *impersonators*, but they are not really *wrappers*.

So, where would I use a wrapper?

We take the view (and this is pure speculation) that a wrapper's intent is to *manage* its content. As part of that function it *may*, but does not have to, include the functions provided by either, or both, the Decorator or the Adapter. A good example arises when it is necessary to integrate entities that are not objects into an object-oriented environment in such a way as to allow the interface of the entity to be addressed as if it really were an object.

For example, a DLL wrapper that provides access to the functions of the DLL can be viewed as acting as an Adapter, but the distinction is that the true wrapper is also responsible for handling basic management functions, which are *not* part of the Adapter pattern. Thus a wrapper will typically include methods for checking that the required DLL is actually available, that it's the correct version, and that it is properly registered. The wrapper is also responsible for releasing the DLL when it is finished with it.

In a purely object-oriented environment, it seems to us that most of the classes that we see referred to as "managers" are really wrappers. For example, a form manager that creates and releases forms, manages a forms collection, and provides an interface for accessing running forms is actually a wrapper. This distinction in name may be because it's dealing with objects but that doesn't alter the intent, which is why we see both as implementations of the same pattern.

Wrapper pattern summary

Table 5 summarizes how we differentiate between the intents of wrappers, Decorators, and Adapters.

***Table 5**. Wrappers, Decorators, and Adapters compared.*

Pattern	Intent
Decorator	Modify behavior without modifying an existing interface
Adapter	Modify interface without modifying existing behavior
Wrapper	Provide interface for, and services to, behavior that is defined elsewhere

Conclusion

In this chapter we have tried to cover what we consider to be the most important, and the most commonly used, patterns for designing for extensibility. However, there are many other well-known and recognized patterns that we have not even mentioned. This is not to say that they are not valuable, or even interesting, but our intention was not to write a comprehensive review of design patterns—there are many others who are far better qualified than us in this area. There are many sources for more information on design patterns, but we do strongly recommend that, if you are interested in the topic, you should read *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides as your starting point for further research.

Chapter 16
VFP on the Web

Visual FoxPro is not, by itself, a tool that can be used to *build* Web applications without some very specialized assistance. There are several tools available that use Visual FoxPro as the basis for sophisticated and powerful Web applications, including West Wind Web Connection and Active FoxPro Pages. However, the issues surrounding the development of such applications are far beyond the scope of this chapter. In this chapter we show you how to use Visual FoxPro to *support* Web applications by data-driving the generation of HTML, creating COM components that can be called from Active Server Pages, and creating and publishing XML Web Services.

How do I data drive the production of HTML?

The combination of Visual FoxPro's powerful string manipulation functions and fast data engine make it particularly good at generating HTML for Web pages. When we consider the task of generating a Web page, there are really three separate issues that we need to address:

- The physical layout of the Web page. A Web page actually consists of a number of individual elements (Title, Header, Body, Footer, and so on) that have to be assembled in a specific order.
- The management of the look and feel of those elements that all of our Web pages share. This involves defining standard fonts and colors for the various elements.
- The management of the content that is to be displayed by each element in the page.

As you can see, the first two are closely related, while the third is actually independent of the others. This is why we have designed two separate classes that cooperate to deliver the functionality required to build a Web page.

How do I give my Web pages a consistent look and feel?

One of the easiest things that you can do to make sure that your Web pages have a similar appearance is to use an external "Cascading Style Sheet" (CSS), which is created as a text file with a .CSS extension. This file specifies design and format information such as the colors, fonts, font sizes, and margins used in the pages of your Web application. The main benefit to using one is that when you want to change the appearance of your entire Web site, you only need to make changes in one file.

Cascading Style Sheets

The simplest way to use a Cascading Style Sheet is to assign standard formatting for each of the HTML tags that you use (or plan to use) in your Web pages. You might specify different fonts for each heading level, a background color for the page body, different colors for your hyperlinks depending on their state, plain text, and so on. So, for example, to ensure that the

body text of all of your Web pages is displayed with black 11 point text on a white background, you simply make an entry like this in your Cascading Style Sheet:

```
body
{
  font-family:verdana, arial, helvetica, sans-serif;
  color:#000000;
  background:#ffffff;
  font-size : 11pt;
}
```

Of course, the simplest way is not the only way! You can also create classes and unique IDs for elements. For example, you might define a class called *AppTitle* in a Cascading Style Sheet like this:

```
.apptitle
{
  font-family:arial, helvetica, sans-serif;
  color:#990033;
  background: #C0C0C0;
  font-size: x-large;
  border : 1px solid #666699;
}
```

Note that the class name must be prefixed with a period, unlike the standard Tag names. It can then be referenced in the HTML used to render the Web page like this:

```
<tr>
  <td class="apptitle" align="center" width="100">
    <img src="gumballs.jpg" align="middle" alt="Application Logo">
  </td>
  <td class="apptitle" valign="top">Generating HTML on the Fly</td>
</tr>
```

The result is the heading shown in **Figure 1**.

In order to apply the styles that you have defined in your Cascading Style Sheet, each HTML document must include a reference that tells it where to find the CSS file. This is done by including a statement like this somewhere within the *head* element of each HTML document:

```
<link rel="stylesheet" type="text/css" href="render.css">
```

Cascading Style Sheet specifications are determined by the World Wide Web Consortium (W3C), the organization that develops common protocols for the World Wide Web. The CSS specification and other information about how Cascading Style Sheets work is available at www.w3.org/Style/ along with information about other style sheet protocols.

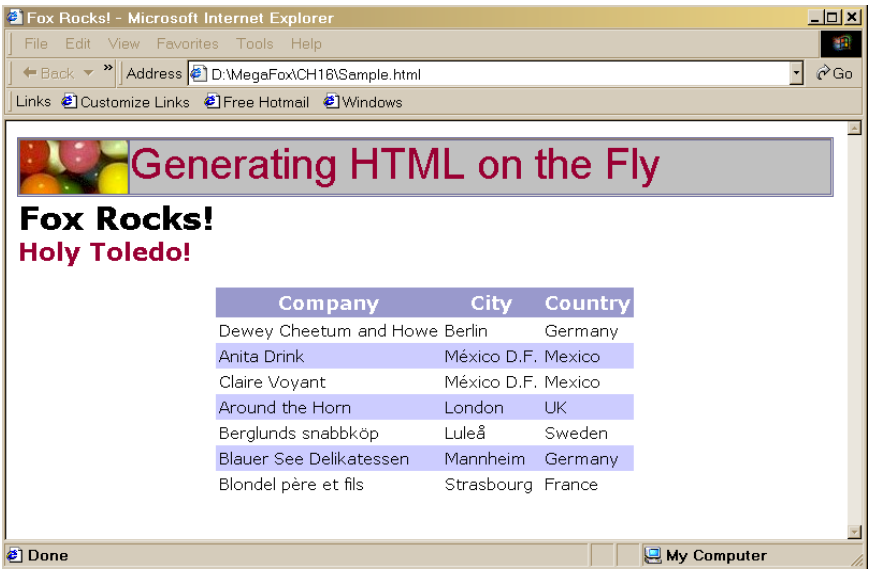


***Figure 1**. To change the formatting, just edit the Cascading Style Sheet.*

Laying out the page (*Example: Render.vcx::Render*)

Cascading Style Sheets are great for defining the appearance of individual parts of a Web page, but they do not address the issue how to actually build the page. While the style sheet defines the color and font for each element, it has nothing to say about the order in which the elements appear. Of course, this can be done explicitly in each HTML document, but then, if you need to change the layout for all of the pages on your Web site you have a lot of work to do! If you were confronted with this problem in a VFP application you would probably tackle it by creating a class to specify and manage the basic layout. So did we; the result is the class we named *Render*!

Before we can set about defining the class, we need to define what the page layout actually is. Our standard layout, which the *Render* class is designed to manage, is illustrated in **Figure 2**.

The class defines a separate property for each segment of our standard page and is intended to work with a Cascading Style Sheet. Each property on the Render object can be mapped to a formatting element defined in the associated style sheet. Those properties that have pre-defined styles have the appropriate Cascading Style Sheet class name listed inside the curly braces.

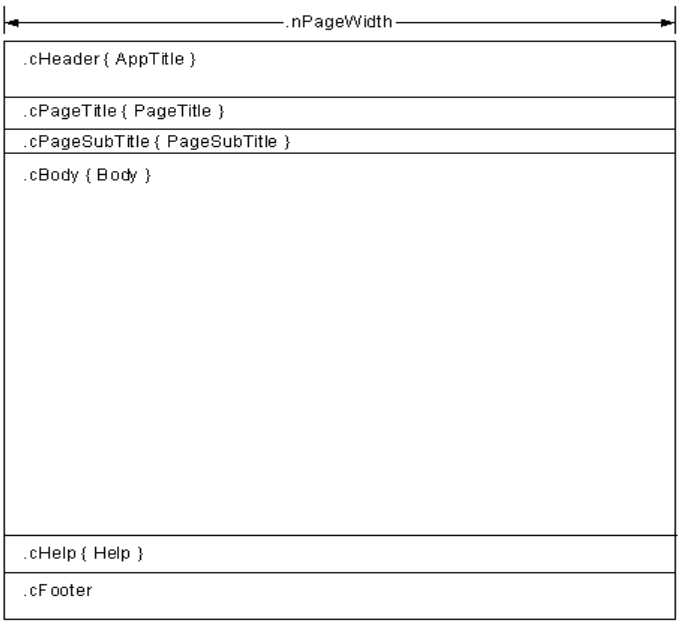


***Figure 2**. The Render class defines the Web page layout.*

How does the Render class work?

The purpose of the class is to construct an HTML page by assembling the data that is stored in its properties in a specific order and adding the relevant tags. The first task, therefore, is to populate these properties, and there are several possible ways of doing this:

- The calling process can populate these properties directly.
- The values can be hard-coded in the *Render* class.
- The values can be read from metadata.

Clearly, many of these properties will contain values that depend on the specific Web page being constructed. In these cases it makes the most sense for the calling process to provide the content. Others, like a standard title block that is used for all the pages on the Web site, can be populated directly by the Render object from metadata because they are essentially static.

We created a custom *PageHeader* class to handle the job of populating the Render's *cHeader* property. The *PageHeader* class has two custom properties that are populated when it is instantiated by the Render object

- *cAppLogo*—Name of an image file to use as the application logo
- *cAppTitle*—Application title

The values used to populate these properties are read from an INI file whose name is stored in the custom *cIniFile* property. Using an INI file to store key pieces of information, such as the application title and associated logo, allows us to customize this class for use with different applications without having to change any code. (Note: We like to use INI files for this sort of task because they are fast and easily understood by even inexperienced developers and users and are therefore easy to maintain and modify. You can, of course, use any appropriate mechanism, perhaps an XML data file or even a Visual FoxPro table.) The INI file used to generate the Web page in Figure 1 looks like this:

```
[config]
Applogo=gumballs.jpg
Apptitle=Generating HTML on the Fly
Bgcolor=#FFFFFF
Stylesheet=render.css
Pagewidth=760
teensybordercolor=#000033
DocTypeTag=<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                "http://www.w3.org/TR/html4/strict.dtd">
```

The PageHeader object has a custom *Render()* method that is called by the Render object when it is ready to construct the Web page. This method constructs the page header by adding the required HTML tags to the content of its *cAppLogo* and *cAppTitle* properties and returns this string to the Render object like this:

```
#DEFINE CRLF CHR( 13 ) + CHR( 10 )
lcRetVal = ""

*** If we have a logo for the application, use it in the page header
IF NOT EMPTY( This.cAppLogo )
  lcRetVal = [<td class="apptitle" align="center" width="100">] + ;
             [<img src="] + This.cAppLogo + [align="middle" ] + ;
             [alt="Application Logo"></td>] + CRLF
ENDIF
*** Add an application title if we have one
IF NOT EMPTY( This.cAppTitle )
  lcRetVal = lcRetVal + ;
    [<td class="apptitle" valign="top">] + This.cAppTitle + [</td>] + CRLF
ENDIF

RETURN lcRetVal
```

The real work of laying out the Web page is done in the Render object's custom *Render()* method. This method surrounds the required content with the necessary HTML tags. Before invoking this method, the *cBody* property must be set to the HTML formatted string that contains the content for the page. This naturally begs the question "How do I generate content for my Web pages?"

How do I generate HTML formatted lists? (*Example: Render.vcx::Lister*)

Despite the apparently infinite variety of styles, shapes, and colors that populate the millions of pages on the World Wide Web, there is one common theme running through the vast majority of them. They are basically mechanisms for displaying lists. Of course, the content

varies widely and pages may include lists of user options, lists of hyperlinks, or lists of products and so on. This is not really surprising when you consider that in order to format a Web page nicely, the easiest way is to format the data as an HTML table. That means rendering it as rows and columns—in other words, lists.

This means that getting data from a table into an HTML formatted list is something that we are going to do over and over again. Rather than keep writing the same (or at least very similar) code to do it, we should encapsulate the basic functionality in a class. Our *Lister* class does exactly that. It is designed to read data from the currently selected alias and transform it into formatted HTML suitable for display on a Web page. If that were all that it did, it would be interesting, but not necessarily riveting. What makes this *Lister* class so special is that it is data driven to provide maximum flexibility.

If you stop and think about it, all lists share a common set of characteristics that define how they are to be formatted. We can model these characteristics in our metadata by defining a field for each. For example, all HTML lists require some sort of prefix, which is implemented as one of the opening tags (`<ul>`, `<ol>`, `<table>`, `<select>`, and so on). This is modeled in the metadata in the column named *cListPrefx*. In addition to the opening tag, all HTML lists require a closing tag, which, in turn, is modeled by the column *cListSuffx*. The resulting table is named LISTER.DBF and its full structure is given in **Table 1**.

***Table 1**. Structure of Lister.dbf.*

Field name	Data type	Field length	Explanation
cID	C	20	Used by the calling program to set the Lister object's ID property. When this property is set, it fires an assign method that locates the appropriate record in the lister table and does the required setup.
cClass	C	20	Can contain the *cID* of another record in the table so that the process controlled by the current row can inherit information contained in a previously defined entry.
cRow	M		Contains an expression that is evaluated at run time to generate the content for the list rows.
cHeader	M		Used to generate the header for the list.
Properties	M		Used to store additional information in the form of one or more attribute/value pairs.
cStyle	M		Used to define visual formatting for the various elements of this type of list (for instance, for a bulleted list, *cStyle* might contain "li, ul {color: black; font-family: verdana, arial, sans-serif; font-size: 9.0pt}").
cListPrefx	C	80	HTML tags to be inserted at the beginning of the list (for instance, for a bulleted list or <table> for a table).
cListSuffx	C	80	HTML tags to be inserted at the end of the list (for instance, for a bulleted list or </table> for a table).
cRowPrefx	C	80	HTML tags to be inserted at the beginning of each row (for instance, for a bulleted list or <tr> for a table).
cRowSuffx	C	80	HTML tags to be inserted at the end of each row (for instance, for a bulleted list or </tr> for a table).

One important thing to note about this design is that we have introduced the concept of a row (which defines a specific list) being related to another row in the same table by the *cClass*

field. The name *cClass* was chosen deliberately because this feature allows lists to inherit from each other. Whenever a row is processed, the contents of the *cClass* field are checked, and if another row has been specified, that row is also processed. This is a classic example of a "reflexive" design.

For example, we might define a plain "vanilla" table in a single row and give it a *cID* of *table*. Then, when we need to define a specific type of table (maybe one with alternating blue and white rows named *AltTable*), instead of having to re-define all the basic table characteristics anew, we simply set the new record's *cClass* field to point to the original *table* entry. Now, when the *AltTable* record is processed, the details from the *table* record will be merged in so that *AltTable* "inherits" the characteristics of *Table*. Only the information that augments what is in the *table* record needs to be specified in *AltTable*. In the case of our table with alternately colored rows, we only need to put the following information into the *cRowPrefix* field:

```
<tr {{IIF(MOD(nListRow,2)=0,[class="evenrow"],[class="oddrow"])}}>
```

And to define the style for even and odd rows in the *cStyle* field like so:

```
.evenrow {background-color:#CCCCFF }
.oddrow {background-color:#FFFFFF}
```

This concept will be familiar if you have ever opened a Visual Class Library and browsed it as a table. It uses a very similar methodology to enable subclasses to inherit from their parents. The class field in a given record contains the *ObjName* of the parent class and the *ClassLoc* field contains the name of the Visual Class Library where the parent class can be found.

How is Lister "inheritance" implemented?

This is actually handled by code in the *Lister*'s *cID_assign* method. Setting the *cID* property to reference a record in LISTER.DBF automatically triggers the location, and evaluation, of that row. A **SCATTER NAME** command is used to store all the data from the record to the *Lister*'s *oConfig* object. The code then walks up the "class tree" that is defined by the references in the *cClass* field of the lister table.

At each level, the information from the required fields of the parent record is merged with the contents of the corresponding property in the *oConfig* object. The process of tree walking begins with the first record and continues as long as the *cClass* field is not empty, as follows:

```
DO WHILE NOT EMPTY( Lister.cClass )
  lcParentClass = LOWER( Lister.cClass )

  *** Find the parent record in the lister table
  IF SEEK( lcParentClass, 'Lister', 'cID' )
    *** Parent class found
    SELECT Lister
```

When a parent record is found all of its fields are processed:

```
    lnCount = AFIELDS( laFields, 'Lister' )
```

```
FOR lnFld = 1 TO lnCount
  lcField = laFields[ lnFld, 1 ]
```

However, the only fields to which inheritance is relevant are *cStyle*, *cListPrefx*, *cListSuffx*, *cRowPrex*, and *cRowSuffx* because the other fields can only ever contain information that relates to the current list anyway. They are simply ignored in this process:

```
IF lcField == 'CID' OR lcField == 'CROW' ;
          OR lcField == 'CHEADER' OR lcField == 'CCLASS'
  LOOP
ENDIF
```

How the remaining fields are processed depends first upon what they contain:

- If empty, there is nothing to do.
- If there is data, and the corresponding property on the *oConfig* object is empty, the property is simply replaced with the contents of the field.
- If both the field and its corresponding property on the *oConfig* object contain data, the two have to be merged.

The first two situations are trivial and easily dealt with:

```
lcParentValue = ALLTRIM( EVALUATE( lcField ) )
lcCurrentValue = ALLTRIM( This.oConfig.&lcField )
DO CASE
  CASE EMPTY( lcParentValue )
    *** Nothing to add or change
    LOOP
  CASE EMPTY( lcCurrentValue )
    *** Straight replacement
    This.oConfig.&lcField = lcParentValue
```

The way in which data is merged depends first upon the field being processed. If it is the *cStyle* field, some fancy footwork is required because in-line styles are formatted as either a tag or a class identifier, followed by a list of attributes and values inside curly braces like this:

```
tag {attribute: value, value, value; attribute: value}
class {attribute: value, value, value; attribute: value}
```

so they must be parsed in order to ensure that they are merged correctly. The *Lister*'s custom *ProcessStyle()* method is called twice; the first time reads the contents of the current record, and the second time adds any existing style definitions from the *oConfig* object:

```
  CASE lcField = "CSTYLE"
    *** Consolidate the styles into one
    IF NOT EMPTY( ALLTRIM( Lister.cStyle ) )
      This.ProcessStyle( Lister.cStyle )
    ENDIF
```

```
IF NOT EMPTY( ALLTRIM( This.oConfig.cStyle ) )
  This.ProcessStyle( This.oConfig.cStyle )
ENDIF
```

It takes information in this form:

```
td {color:black;font-family:verdana,arial,sans-serif}
```

and writes it out to a cursor named *StyleSummary* that has three fields: one for the identifier (tag or class), one for the attribute, and one for the associated list of values like this:

```
cTag    cAttrib         cValue
td      color           black
td      font-family     verdana,arial,sans-serif
```

The cursor is indexed on *cTag* + *cAttrib* and because the current record (that is, the parent) is always processed first, any duplicate combinations are overwritten by the current contents of *oConfig*, thereby providing the ability to override inherited styles sequentially. The composite in-line style statement is then constructed for each identifier by simply scanning the cursor:

```
lcAll = ""
IF USED( 'StyleSummary' )
  SELECT StyleSummary
  GO TOP
  DO WHILE NOT EOF()
    lcRefTag = cTag
    lcOut = "{"
    SCAN WHILE cTag = lcRefTag
      lcOut = lcOut + ALLTRIM( cAttrib ) + ":" + ;
                      ALLTRIM( cValue ) + CHR( 59 ) && SemiColon
    ENDSCAN
    lcAll = lcAll + ALLTRIM( lcRefTag ) + " " + lcOut + "}" + CRLF
  ENDDO
  USE IN StyleSummary
  SELECT Lister
ENDIF
This.oConfig.cStyle = lcAll
```

The final contingency that must be accounted for is that we are processing one of the prefix or suffix fields, and there are two distinct scenarios that can apply. The first is that the tag is specified in the parent record and the child record merely modifies that tag with additional attributes. For example, the *cListPrefx* field in the parent record may contain **`<table>`**, the first child record **`width="100%"`**, and the next **`border=1`**. These need to be merged to produce:

```
<table width="100%" border=1>
```

The second case is that the child record specifies the entire tag. In this case we need to override anything from the "parent" record. The first step in processing these tags is to create a pair of two-dimensional arrays: one for the attributes contained in the *oConfig* object and one

for the information in the parent record. The first column of each array contains whatever is on the left side of the equal sign (the attribute) and the second column contains whatever is on the right (the value).

```
CASE LEFT( lcCurrentValue, 1 ) # '<'
  lcTag = ""
  *** What we have now is not a tag, so add/replace it as an attribute
  *** Make an array of current values. This code takes into account the
  *** situation where we may have multipe attributes in a single field
  *** with each attribute-value pair separated by a space
  FOR lni = 1 TO GETWORDCOUNT( lcCurrentValue )
    DIMENSION laCurrentAttributes[ lni, 2 ]
    lcCurrentExp = GETWORDNUM( lcCurrentValue, lni )
    laCurrentAttributes[ lni, 1 ] = GETWORDNUM( lcCurrentExp, 1, '=' )
    laCurrentAttributes[ lni, 2 ] = GETWORDNUM( lcCurrentExp, 2, '=' )
  ENDFOR
  lnAdjust = 0

  *** Make an array of parent class values
  FOR lni = 1 TO GETWORDCOUNT( lcParentValue )
    lcParentExp = GETWORDNUM( lcParentValue, lni )
    IF lni = 1 AND LEFT( lcParentExp, 1 ) = '<'
      *** If we have a tag here, we do not want to include
      *** it in the array, the array contains only attributes
      lcTag = STRTRAN( STRTRAN( ;
                GETWORDNUM( lcParentExp, 1 ), '<' ), '>' )
      lnAdjust = 1
      LOOP
    ENDIF
    DIMENSION laParentAttributes[ lni - lnAdjust, 2 ]
    laParentAttributes[ lni - lnAdjust, 1 ] = ;
      GETWORDNUM( lcParentExp, 1, '=>' )
    laParentAttributes[ lni - lnAdjust, 2 ] = ;
      GETWORDNUM( lcParentExp, 2, '=>' )
  ENDFOR
```

When the two arrays have been built, they are merged together and, if the same attribute appears in both arrays, the one from the parent array is discarded. The result is that the information specified in the child overrides that from the parent!

```
  FOR lni = 1 TO ALEN( laCurrentAttributes, 1 )
    *** case insensitive search searching on column 1
    *** with EXACT set on and return the row number
    lnAtPos = ASCAN( laParentAttributes, ;
                     laCurrentAttributes[ lni, 1 ], -1, -1, 1, 15 )
    IF lnAtPos = 0
      *** Augment
      lnIndex = ALEN( laParentAttributes, 1 ) + 1
      DIMENSION laParentAttributes[ lnIndex, 2 ]
      laParentAttributes[ lnIndex, 1 ] = laCurrentAttributes[ lni, 1 ]
      laParentAttributes[ lnIndex, 2 ] = laCurrentAttributes[ lni, 2 ]
    ELSE
      *** Over-ride
      laParentAttributes[ lnAtPos, 2 ] = laCurrentAttributes[ lni,2 ]
    ENDIF
  ENDFOR
```

The last step in the process is to add back the tag itself, together with the required opening and closing characters:

```
          IF NOT EMPTY( lcTag )
            lcAll = "<" + lcTag
          ELSE
            lcAll= ""
          ENDIF
          FOR lni = 1 TO ALEN( laParentAttributes, 1 )
            IF EMPTY( laParentAttributes[ lni, 1 ] )
              LOOP
            ENDIF
            lcAll= lcALL + " " + laParentAttributes[ lni, 1 ]
            IF LEN( laParentAttributes[ lni, 2 ] ) > 0
              lcAll = lcAll + [=] + laParentAttributes[ lni, 2 ]
            ENDIF
          ENDFOR
          IF NOT EMPTY( lcTag)
            lcAll = lcAll + ">"
          ENDIF
          This.oConfig.&lcField = lcAll
      ENDCASE
    ENDFOR  && FOR lnFld = 1 TO lnCount
  ELSE
    ASSERT .F. MESSAGE lcParentClass + ' NOT FOUND in Lister.dbf'
  ENDIF  && IF SEEK( lcParentClass, 'Lister', 'cID' )

 ENDDO  && DO WHILE NOT EMPTY( Lister.cClass )
```

As you can see, implementing this reflexive table requires quite a lot of code. The good news is that it only has to be written once, and it is already done.

How is the content of the list generated?

Remember, the objective here is to generate the HTML string that is used to populate the *cBody* property on the Render object. The calling process must first ensure that the cursor whose contents are to be displayed in the list is open in the currently selected work area. Then it must create an instance of the Lister object. Next it must set the object's *cID* property to the *cID* of the required row in LISTER.DBF to process the metadata. Finally, the HTML string is actually generated by calling the Lister object's *Execute()* method.

```
SELECT <data source>
loLister = CreateObject( 'Lister' )
loLister.cID = <Lister.cID>
lcOutput = loLister.Execute()
```

The HTML string is built up in stages, and the result for each stage is stored to a different property on the Lister object and then assembled at the very end. The three properties are:

- *cRecords* The HTML representation of data from the source table
- *cHeader* The list's HTML preamble, which includes the in-line styles, opening tags, and header row (where specified)

- *cFooter* Anything that is to appear on the page after the list (optional)

You will recall that the code in the *cID_assign()* method did not process either the *cHeader* or *cRow* fields, and we stated that this was because these fields never inherit from other records in the metadata. The reason is that the *cHeader* field is used to define the content for the list's header, while *cRow* defines the content for the body of the list. Each is, of course, specific to the current list. **Table 2** shows the contents of the metadata that is used to generate the list shown in Figure 1.

***Table 2**. Metadata used to generate the client list.*

Field name	Field value
cID	clients
cClass	alttable
cRow	*** Company name first <td>{{ ALLTRIM(cCompany) }}</td> *** Now city <td>{{ ALLTRIM(cCity) }}</td> *** And finally country <td>{{ ALLTRIM(cCountry) }}</td>
cHeader	<th>{{DBGETPROP('Clients.cCompany', 'Field', 'Caption')}}</th> <th>{{DBGETPROP('Clients.cCity', 'Field', 'Caption')}}</th> <th>{{DBGETPROP('Clients.cCountry', 'Field', 'Caption')}}</th>

This code, in the *Lister*'s custom *Execute()* method, scans the records in the currently selected alias and calls its custom *OnRecord()* method to apply the contents of the *cRow* field to each record in the cursor to generate the content.

```
*** Must be private so that it is in scope
*** when oConfig.cStyle is processed
pnListRow = 0

*** If we have no records in the cursor in the selected
*** work area, there is nothing to render
IF RECCOUNT() > 0
  SCAN
  pnListRow = pnListRow+ 1
  This.cRecords = This.cRecords + This.OnRecord()
ENDSCAN
```

OnRecord() uses the **TEXTMERGE()** function to insert the information from the current record into the current row of the HTML list. However, evaluating what is in the current record is only part of the job. This method is also responsible for inserting any tags that are required at the beginning and end of the row.

```
RETURN TEXTMERGE( ALLTRIM( This.oConfig.cRowPrefx ), .T., '{{', '}}' ) + ;
       TEXTMERGE( ALLTRIM( This.oConfig.cRow ), .T., '{{', '}}' ) + ;
       TEXTMERGE( ALLTRIM( This.oConfig.cRowSuffx ), .T., '{{', '}}' ) + CRLF
```

The preamble to the list is generated by invoking the *Lister*'s *OnHeader()* method. This method concatenates the components that it retrieves from the *oConfig* object and stores the result to the *Lister*'s *cHeader* property:

```
*** Generate the preamble to the list
*** Typically the <table> statement and the
*** column header row if it's specified.
LOCAL lcRetVal
#DEFINE CRLF CHR( 13 ) + CHR( 10 )
lcRetVal = CRLF
IF NOT EMPTY( This.oConfig.cStyle )
  lcRetVal= lcRetVal + [<style>] + CRLF + ;
        TEXTMERGE( ALLTRIM( This.oConfig.Style ), .T., '{{', '}}' ) + CRLF + ;
            [</style>] + CRLF
ENDIF
lcRetVal = lcRetVal + ;
   TEXTMERGE( ALLTRIM( This.oConfig.cListprefx ), .T., '{{', '}}' ) + CRLF

IF NOT EMPTY( This.oConfig.cHeader )
  lcRetVal = lcRetVal + ;
        TEXTMERGE( ALLTRIM( This.oConfig.cRowPrefx ), .T., '{{', '}}' ) + ;
        TEXTMERGE( ALLTRIM( This.oConfig.cHeader ), .T., '{{', '}}' ) + ;
        TEXTMERGE( ALLTRIM( This.oConfig.cRowSuffx ), .T., '{{', '}}' ) + CRLF
ENDIF
RETURN lcRetVal
```

Similar code in the custom *OnFooter()* method generates the HTML for the list suffix (including the closing tag) and any footer information and stores it to the *Lister*'s *cHeader* property. Finally, the contents of the three properties are concatenated and passed back to the caller as an HTML formatted list.

Putting it all together (*Example: GenHTML.pjx and GenerateHTML.scx*)

The sample code provided with this chapter demonstrates how you can use the *Render* and *Lister* classes to generate HTML on the fly. There are two ways of using the examples—the first is to run the sample form GENERATEHTML.SCX. This form uses the *Render* and *Lister* classes to generate an HTML stream, saves it to a file, and displays it in the _WebBrowser4 control that ships with the VFP foundation classes. The second way is to compile the GENHTML.PJX project as a DLL and call the *GenPage()* method of the *SesGenPage* class from an Active Server Page to return an HTML stream suitable for display as a Web page.

Displaying the Web page in Visual FoxPro

The sample form GENERATEHTML.SCX (see **Figure 3**) uses the *Render* and *Lister* classes directly in the custom *GenerateHTML()* method of the form.

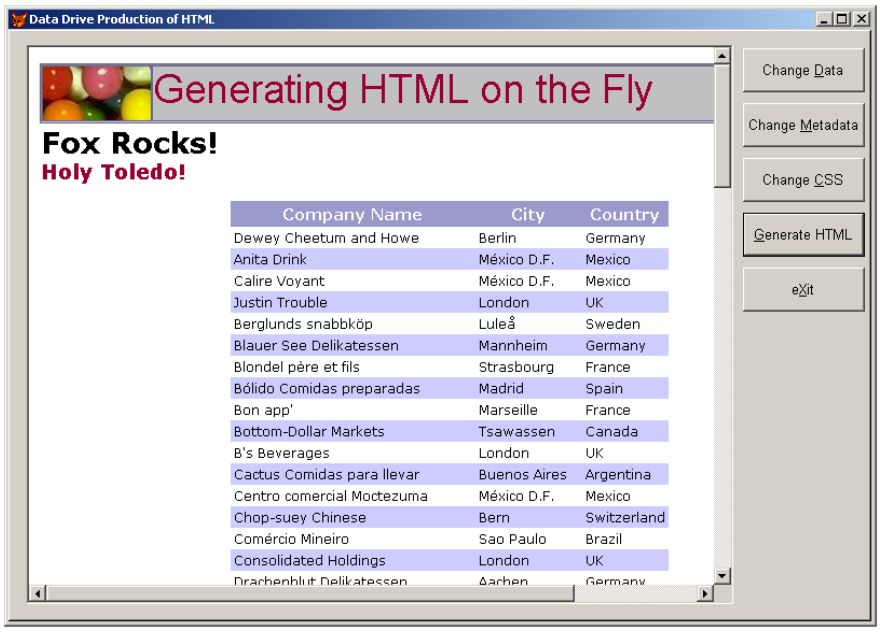


***Figure 3**. A Visual FoxPro HTML test bed.*

However, because of the way that the *Render* and *Lister* classes are designed, surprisingly little instance-level code is required to render the HTML page. This is all that is needed to get the job done:

```
LOCAL lcHtml
*** Select and position the clients table
SELECT Clients
GO TOP
WITH Thisform.oRender
  *** Reset the render object
  *** in case the cascading style sheet has been modified
  *** so he will re-read it and see the changes
  .Reset()
  *** And set up a pagetitle
  .cPageTitle = 'Fox Rocks!'
  .cPageSubTitle = 'Holy Toledo!'
  *** Set the lister up
  .oLister.cID = 'clients'
  .cBody = .oLister.Execute()
  *** This produces the html file
  lcHtml = .Render()
  IF STRTOFILE( lcHtml, 'Sample.html' ) < 1
    MESSAGEBOX( 'Unable to create HTML file', 16, 'Major WAAAAHHHHH!' )
  ENDIF
ENDWITH

Thisform.pgfRender.pgRender.oBrowser.Navigate2 ;
  ( FULLPATH( CURDIR() ) + 'Sample.html' )
```

This form is much more than a simple demonstrator for the output of the *Render* and *Lister* classes. It also allows you to test any changes to the underlying data, the metadata that is used by the *Lister* class and the Cascading Style Sheet. (Remember to hit the Generate HTML button after making any change in order to see the results.) Moreover, since the code in *Render* and *Lister* is data driven, it is not limited to working solely with the sample data, and with very few changes this form could be used with any Visual FoxPro data source as a generic test bed for developing list-based Web pages.

Using the DLL from an Active Server Page

To see how these classes work when they are implemented in Active Server Pages, just follow these steps:

1. Build a multi-threaded DLL from GENHTML.PJX and name it GENHTML.DLL (only because this name is hard-coded into the ASP page!).
2. Open the Internet Services Manager and create a virtual directory under "default Web sites" and point it to the folder with the sample code for this chapter.
3. Open Internet Explorer and navigate to *Localhost/<your virtual directory name>*.

The Active Server Page (which is named PAGESELECT.ASP) requires almost no code to generate the Web page. This is all that is required to call upon the VFP components to generate and return the requested HTML:

```
Set oPageGen = CreateObject( "GenHTML.SesGenPage" )
cHTML = oPageGen.GenPage( "Clients" )
Response.Write( cHTML )
```

The preceding code instantiates the *sesGenPage* class that exposes a method named *GenPage()*, which, when called with a valid LISTER.DBF *cID* value, returns an HTML string that is ready for display as a Web page.

An ancillary benefit to the data-driven design we have adopted is that changes can be made to both the appearance and content of Web pages merely by changing the data in LISTER.DBF or by modifying the Cascading Style Sheet. There is no need to recompile the application, which means, of course, that there is no need to take down a Web server to make minor changes.

What are the Office Web Components?

The Office Web Components are a set of COM controls that were first introduced in Microsoft Office 2000. They allow you to add interactivity to Excel charts, spreadsheets, and pivot tables saved in HTML format. All the controls support a rich set of programming interfaces that you can call from any language capable of calling a dual or dispatch COM interface. The Office Web Components were updated for Microsoft Office XP, with a new installation and a new view-only mode so that even users who have not installed Office XP can still view data. This is a major advantage over the Office 2000 version that requires that all client machines have Office 2000 installed.

Unfortunately, some pretty fundamental changes were made to the object model when the components were updated. There is no guarantee that code written using the Office Web Components for Office 2000 will continue to run if you upgrade to the XP version. The only advice we can offer is to try it on a test machine before putting anything into production.

In Office XP, the Office Web Components can be installed either as part of the Office XP installation or separately. When you install both Office XP and the Office Web Components, you get data interactivity on your Web pages. For example, you can add or update data in a spreadsheet, rearrange columns in a pivot table, or alter a chart. When you install the Office Web Components without Office XP, you are limited to view-only mode for the data. You can still view data on the pages, and print the pages, but you cannot interact.

How do I install the Office Web Components?

The newest version of the Office Web Components can be installed in one of three ways:

- Installing Office XP installs the Office Web Components by default.
- Using the additional SETUP.EXE file in the \File\owc folder of the Office XP CD allows you to install just the Office Web Components.
- The Office Web Components are also available as a self-extracting downloadable file from Microsoft at http://office.microsoft.com/downloads/2002/owc10.aspx. This is useful when you want clients that do not have Office XP installed to have read-only access to applications that are built using these components.

The Office Web Components have the same system requirements as the rest of Office XP. They are supported by Internet Explorer 4.01 or later, running on Microsoft Windows 98, Windows NT 4.0, Windows 2000 and later.

> ! *Note: Because they rely on ActiveX technology, HTML documents that contain Office Web Components do not, at the time of writing, run in any Web browser other than recent versions of Internet Explorer for Windows.*

How do I create graphs using the Office Web Components?

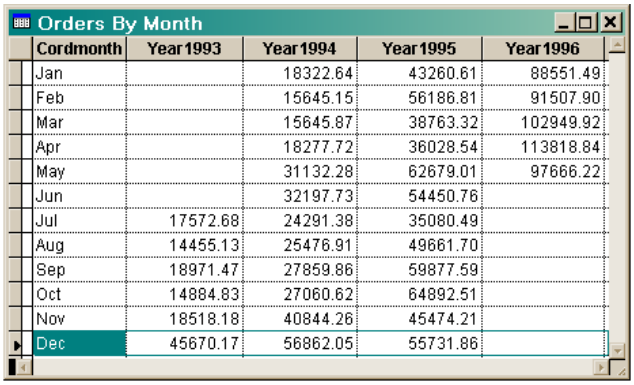
There are at least two ways to display graphs in a Web page. We can manipulate the chart object in Visual FoxPro just as we did when we created graphs using Excel Automation (see Chapter 6) and use it to create a temporary GIF file. Alternatively, we can actually embed the chart object in a Web page between `<Object>` tags and generate either VBScript or JavaScript to format the graph when the browser window loads. The second approach has the advantage that we do not need to worry about cleaning up the "garbage" files that accumulate when we save charts as temporary files.

How do I save a chart as a GIF file?

It is relatively easy to create a chart object and feed it some data to generate a graph. We can then manipulate the graph's visual properties to format it nicely before exporting it to a GIF file. This file can then be displayed in a Web page between `<Img>` tags or in a Visual FoxPro form by setting the *Picture* property of an Image control to point to the newly created file.

Let us supposed that we want to generate a graph for data that looks like the cursor in **Figure 4**. The process is fairly straightforward using the Office Web Components for Office XP. First, we must create an instance of the ChartSpace object. (This is one of the things that changed in the Office XP version. In Office 2000, we must create an instance of the Chart object instead.)

```
*** If you are using an earlier version of the owc,
*** this will be 'owc.chart'
loChartSpace = CREATEOBJECT( 'owc10.ChartSpace' )
```



Orders By Month

Cordmonth	Year1993	Year1994	Year1995	Year1996
Jan		18322.64	43260.61	88551.49
Feb		15645.15	56186.81	91507.90
Mar		15645.87	38763.32	102949.92
Apr		18277.72	36028.54	113818.84
May		31132.28	62679.01	97666.22
Jun		32197.73	54450.76	
Jul	17572.68	24291.38	35080.49	
Aug	14455.13	25476.91	49661.70	
Sep	18971.47	27859.86	59877.59	
Oct	14884.83	27060.62	64892.51	
Nov	18518.18	40844.26	45474.21	
Dec	45670.17	56862.05	55731.86	

***Figure 4**. Data from csrResults used to generate the graph.*

Because the Office Web Components were designed to work in Web pages, and named constants cannot be used in VBScript (VBScript regards the named constant as just another uninitialized variable), Microsoft conveniently supplied a mechanism for using named constants with the Office Web Components. The top-level container objects (ChartSpace, DataSourceControl, PivotTable, and Spreadsheet) all expose a *Constants* property. This is an object that contains all of the named constants available in the Microsoft Office Web Components type library. To use these named constants, all we need to do is qualify them in our code like this:

```
loConstants = loChartSpace.Constants
```

The next step in generating the graph is to clear any existing graphs before we add a chart to our ChartSpace. The ChartSpace is the container for charts and is able to contain more than one of them.

```
loChartSpace.Clear()
loChart = loChartSpace.Charts.Add()
```

Before we process the data, we format the chart as clustered column and give it a legend like this:

```
loChart.Type = loConstants.chChartTypeColumnClustered
```

```
loChart.HasLegend = .T.
```

Next we need to process the cursor. The first thing that needs to be done is to extract the data from the first column of the cursor (*csrResults*) into a one-dimensional array. This array is used to generate the category labels.

```
  lcFieldName = FIELD( 1, 'csrResults' )
  SELECT &lcFieldName FROM csrResults INTO ARRAY laLabels
  *** Now redimension the array so it is one-dimensional
  lnLen = ALEN( laLabels, 1 )
  DIMENSION laLabels [ lnLen ]
```

The easiest way to generate the graph is to add a Series object for each column of data. Then call its *SetData()* method, passing a one-dimensional array of the values that are to be used to define the data points.

```
lnFields = FCOUNT( 'csrResults' )
WITH loChart
  FOR lnSeries = 2 TO lnFields

    *** Find out the name of the field associated with this series
    lcFieldName = FIELD( lnSeries, 'csrResults' )

    *** Get the data from this column of the cursor into an array
    SELECT &lcFieldName FROM csrResults INTO ARRAY laSeriesData

    *** Now redimension the array so it is one-dimensional
    lnLen = ALEN( laSeriesData, 1 )
    DIMENSION laSeriesData [ lnLen ]

    *** Add the series object
    loSeries = .SeriesCollection.Add()

    *** And set its properties
    WITH loSeries
      .Caption = STRTRAN( PROPER( FIELD( lnSeries, 'csrResults' ) ), ;
                          'Year', '', -1, -1, 1 )
      .SetData( loConstants.chDimCategories, ;
                loConstants.chDataLiteral, @laLabels )
      .SetData( loConstants.chDimValues, ;
                loConstants.chDataLiteral, @laSeriesData )
    ENDWITH
  ENDFOR
ENDWITH
```

Finally, we can format the graph before exporting it to the GIF file by manipulating its many properties. For a complete list of the properties and methods of the chart object, refer to the Help file for the Office Web Components (OWCVBA10.CHM). For example, the following code displays the legend at the bottom of the chart and changes its font size:

```
loChart.Legend.Position = loConstants.chLegendPositionBottom
loChart.Legend.Font.Size = 8
```

We can also set titles and labels (both font and orientation) for the category axis like this:

```
WITH loChart.Axes( loConstants.chAxisPositionCategory )
  .HasTickLabels = .T.
  .Orientation = loConstants.chLabelOrientationHorizontal
  .Font.Size = 8
  .HasTitle = .T.
  .Title.Caption =  "This is the Category Axis"
  .Title.Font.Size = 10
ENDWITH
```

Once the graph is formatted to our liking, all that is left is to create the GIF file:

```
lcFileName = SYS( 2015 ) + '.gif'
loChartSpace.ExportPicture( FULLPATH(CURDIR()) + lcFileName, 'gif', 750, 480 )
```

When we originally started writing the sample code for this chapter in January 2002, we were going to design a class to encapsulate this process and hide the complexity. As luck would have it, around that time, Rick Strahl wrote a paper describing just such a class that wraps the functionality of the OWC Chart control. You can download his paper and the class from here: www.west-wind.com/presentations/OWCCharting/OWCCharting.asp.

How do I use the chart as an embedded object in my Web page? (*Example: Render.vcx::OwcGraph and OwcChartDemo.scx*)
One of the main problems with generating graphs and charts as temporary GIF files is deciding how and when to clean them up. An alternative is to embed the chart directly in the page. However, this does require that the clients have the Office Web Components installed on their local machines. They are downloadable free of charge so cost is not an issue, but what may be an issue, particularly if your application is distributed outside your immediate organization, is that many companies have strict rules about what gets installed on their systems—especially in the context of ActiveX Web controls. So, while this solution, may make your life easier, it may be wise to check with your clients and users before adopting it.

To create a ChartSpace object directly on a Web page, use this syntax:

```
<object id=oChartSpace classid=CLSID:0002E556-0000-0000-C000-000000000046>
```

Notice that this requires the ChartSpace Class ID to be embedded explicitly. You can find the Class ID for each of the Office Web Components in their individual entries in the Help file (OWCVBA10.CHM), but for convenience they are also listed in **Table 3**.

***Table 3**. Office Web Component Class IDs.*

Object	Class ID
ChartSpace	CLSID:0002E556-0000-0000-C000-000000000046
RecordNavigationControl	CLSID:0002E554-0000-0000-C000-000000000046
DataSourceControl	CLSID:0002E553-0000-0000-C000-000000000046
PivotTable	CLSID:0002E552-0000-0000-C000-000000000046
Spreadsheet	CLSID:0002E551-0000-0000-C000-000000000046

Embedding chart objects directly has a couple of benefits. First, there are no temporary files to clean up. Second, when users want to change their view of a particular chart, there is no need for a round trip to the server, or to generate another file; it can be handled locally in the Web page.

In Chapter 6, "Creating Charts and Graphs," we introduced a data-driven methodology for obtaining the data used to generate are graphs. We are using the same methodology here—so remember that the data source used for the graph must always be a cursor named *csrResults*, which uses the first column to define the category labels. The important part of our code here is in the custom *GenerateScript()* method of the *OwcGraph* class. It is called by the *CreateGraph()* method to return a string that contains the scripts needed to render the graph in the Web page. The code may also look familiar to you because it is very similar to the code we used to manipulate the graph object in the preceding section.

The first thing that the *GenerateScript()* method does is to get the static part of the required script, which is stored in the file named GRAPHTYPES.TXT. In the example, this includes all the HTML required to display the list of chart types, and the script required to modify the appearance of the chart when the user selects a different chart type. It also includes the line of code we showed earlier to initialize the ChartSpace object.

```
*** Generate the beginning of the script
lcScript = FILETOSTR( 'GraphTypes.txt' ) + CRLF
```

The next part of the script is generated dynamically from *csrResults*. Each column of data in the cursor (other than the first, which defines labels) is used to populate a Chart series object, and the row count defines the number of categories that will be required in the chart. Instead of simply creating the category array, as in the previous example, we first generate the VBScript that is required to create and dimension the same array in the Web page.

```
*** get the number of fields in the cursor
lnFields = FCOUNT( 'csrResults' )

*** and the number of categories
lnCategories = RECCOUNT( 'csrResults' )

*** Now set up the categories array
*** This is the first field in the cursor
lcScript = lcScript + CRLF + ;
             'Dim aCategories( ' + TRANSFORM( lnCategories ) + ' )' + CRLF

*** Get the labels from the first column of the cursor into an array
lcFieldName = FIELD( 1, 'csrResults' )
SELECT &lcFieldName FROM csrResults INTO ARRAY laLabels
lnLen = ALEN( laLabels, 1 )
FOR lnCnt = 1 TO lnLen
```

```
  *** In VBScript, the array is 0-based, so adjust the index
  lcScript = lcScript + ;
              [aCategories(] + TRANSFORM( lnCnt - 1 ) + [) = "] + ;
              lalabels[ lnCnt ] + ["] + CRLF
ENDFOR
```

Next, we need to generate the VBScript that will create and populate the arrays used to fill the series objects.

```
lcScript = lcScript + CRLF + ;
             'Dim aValues( ' + TRANSFORM( lnCategories ) + ' )' + CRLF
FOR lnSeries = 2 TO lnFields

  *** Find out the name of the field associated with this series
  lcFieldName = FIELD( lnSeries, 'csrResults' )

  *** Get the data from this column of the cursor into an array
  SELECT &lcFieldName FROM csrResults INTO ARRAY laSeriesData
  FOR lnCnt = 1 TO lnLen

    *** In VBScript, the array is 0-based, so adjust the index
    lcScript = lcScript + ;
                [aValues(] + TRANSFORM( lnCnt - 1 ) + [) = ] + ;
                   TRANSFORM( laSeriesData[ lnCnt ] ) + CRLF
  ENDFOR
  lcScript = lcScript + [Set oSeries = oChart.SeriesCollection.Add] + CRLF
  lcScript = lcScript + [oSeries.Caption = "] + ;
               STRTRAN( PROPER( FIELD( lnSeries, 'csrResults' ) ), ;
                         'Year', '', -1, -1, 1 ) + ["] + CRLF
  lcScript = lcScript + ;
               [oSeries.SetData oConstants.chDimCategories, ] +;
               {oConstants.chDataLiteral, aCategories] + CRLF
  lcScript = lcScript + [oSeries.SetData oConstants.chDimValues,] +;
              [oConstants.chDataLiteral, aValues] + CRLF
ENDFOR
```

Finally, the necessary ending and close tags are added and the entire HTML string is returned to the calling process.

```
lcScript = lcScript + [End Sub] + CRLF + [</script>] + CRLF
lcScript = TEXTMERGE( lcScript, .T., '{{', '}}' )
RETURN lcScript
```

If the calling process is an Active Server Page, as in this example, the returned string would then be used in a call to *Response.Write()* in order to generate the page (see **Figure 5**). In a Visual FoxPro Form the string is simply saved to a file for display in a browser control.

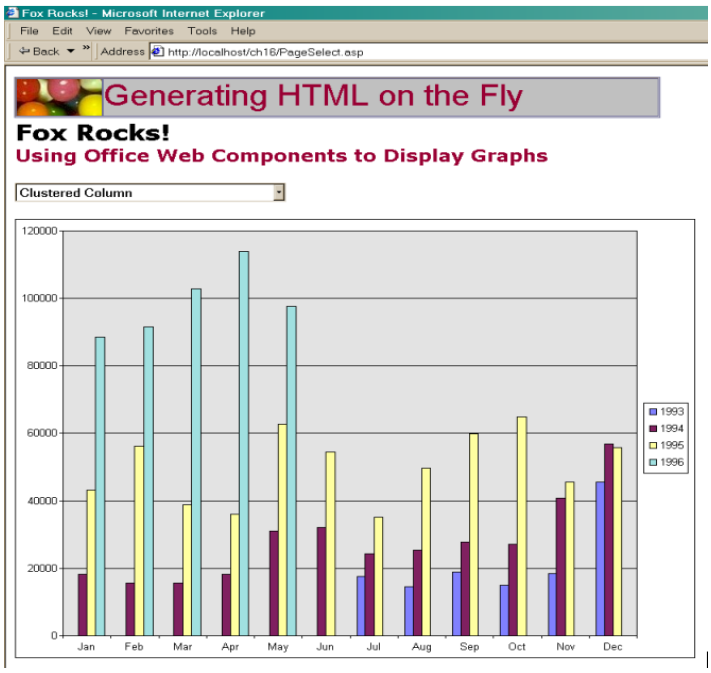


***Figure 5**. Chart object displayed in Web page.*

How do I keep from having to take my Web server down when I modify my DLL? (*Example: GenHTMLbyProxy.prg and VfpProxy.prg*)

One of the problems that we encounter when developing applications for the World Wide Web is that once a COM server has been instantiated in an ASP page, Internet Information Server (IIS) locks it into memory to improve future performance. This means that, in order to update the COM server, we have to momentarily shut down IIS. This can be a problem when we are talking about live Web sites that require constant tuning and modification. Fortunately, there is a workaround for this: We can create a proxy DLL that passes the call onto the class that does the real work.

The idea here is that the Active Server Page instantiates a proxy DLL whose only function is to be grabbed and held by IIS. The proxy exposes a single method that is called from the Active Server Page, with parameters indicating the name, location, and any required parameters of the real functionality. Since the proxy DLL has no real functionality, we do not need to worry about updating it, so we avoid having to take our Web server down.

As a side benefit, the actual functionality no longer needs to be compiled as a DLL either. Providing that it is accessible to the proxy it may, for example, be left as a simple PRG file. There may also be a small performance penalty to pay for adopting this approach (remember, there's no such thing as a free lunch!). You just have to assess each situation on its merits.

If you have already performed Steps 1-3 in the section entitled "Using the DLL from an Active Server Page," only a few modifications are required to see the proxy in action:

1. Build a multi-threaded DLL from PROXY.PJX and name it PROXY.DLL (only because this name is hard-coded into the ASP page!).
2. Modify DEFAULT.ASP to call PGSELBYPROXY.ASP instead of PAGESELECT.ASP by changing this line:

```
<form name="SelectPage" method="POST" action="PgSelByProxy.asp">
```

The code in the Active Server Page example (PGSELBYPROXY.ASP) shows how a proxy is used to call the same functionality that we used in the previous section to generate an HTML list or graph. We begin by instantiating the proxy and invoking its *Execute()* method, which makes the call to the specified function, in this case *PassItOn* in the program GENHTMLBYPROXY.PRG.

```
Set oProxy = Server.CreateObject( "Proxy.VFPProxy" )
IF cPageID = "Display Client List" Then
  cHTML = oProxy.Execute( "PassItOn( [GenPage( 'Clients' )] )",_
                          "GenHTMLByProxy.prg" )
Else
  cHTML = oProxy.Execute( "PassItOn( [GenGraph( 'MonthlySales' )] )",_
                          "GenHTMLByProxy.prg" )
End If
```

The proxy DLL exposes a single *olePublic* class named *VFPproxy*. *VFPproxy*'s custom *Execute()* method is based on the assumption that the required functionality has been defined as a function or procedure contained in a PRG file. If, as in our example, we need to instantiate a class and call its methods, the target function must handle it.

```
lcFile = This.cCurDir + tcFile
IF '.PRG' $ UPPER( lcFile )
  SET PROCEDURE TO &lcFile
  luRetVal = &tcCommand
  SET PROCEDURE TO
  CLEAR PROGRAM &lcFile
  RETURN luRetVal
```

The *PassItOn()* function in GENHTMLBYPROXY.PRG creates the *sesGenPage* object, calls the appropriate method, and returns the generated HTML to the proxy. The proxy then returns the result to the ASP page that instantiated it.

```
FUNCTION PassItOn( tcParms )
LOCAL loPageGen, lcHTML, lcCmd
*** Create an  instance of the HTML generator
loPageGen = NEWOBJECT( 'SesGenPage', 'GenerateHTML.prg' )
*** Pass it on
IF VARTYPE( loPageGen ) = 'O'
  *** Construct the method call
  lcCmd = 'loPageGen.' + tcParms
  lcHTML = &lcCmd
```

```
ELSE
  lcHTML = FILETOSTR( 'Error.txt' )
ENDIF
RETURN lcHTML
```

You will recall that in our original design, this method was part of our OLE public class and it was instantiated directly by the ASP page. By interposing the proxy between the ASP page and the HTML generator, we can now make changes to the way we generate HTML without having to restart IIS. It does require a little more code, and makes our calls in the ASP pages a little less obvious, but in our opinion this is a small price to pay for the benefit that we obtain.

How do I publish a Web Service?

Chapter 5 introduced Web Services, and we showed you how easy it is to consume them in Visual FoxPro 7.0. Publishing them requires a little more work, but Visual FoxPro provides us with wizards to make this task a lot easier. In this chapter we show you how to create and publish a Visual FoxPro Web Service and explain some of the terminology that surrounds them.

The first step in building a Web Service is to build a Visual FoxPro multi-threaded DLL (see Chapter 14) that includes the methods that you want the Web Service to expose over the Internet (or intranet). After you have built the DLL, right-click on the main program in the Project Manager and select "Builder" from the context-sensitive menu (see **Figure 6**).

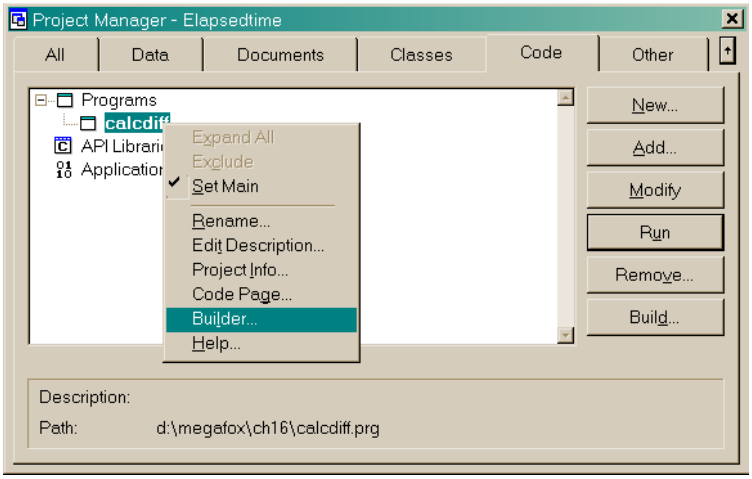


***Figure 6**. Step 1: Select "Builder" from the context-sensitive menu.*

This displays the Wizard Selection dialog (see **Figure 7**). A new entry in Visual FoxPro 7.0 is the "Web Services Publisher."

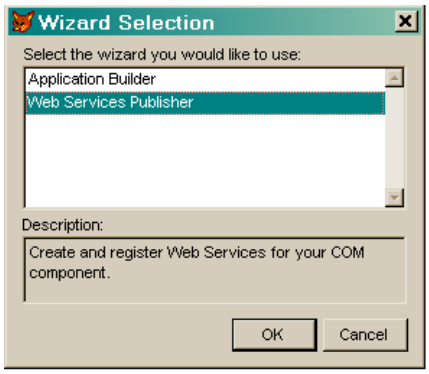


***Figure 7**. Step 2: Select Web Services Publisher from the Wizard Selection dialog.*

Choosing the Web Service Publishing wizard displays the Visual FoxPro Web Services Publisher dialog shown in **Figure 8**. By default, this dialog appears in its "closed" form.

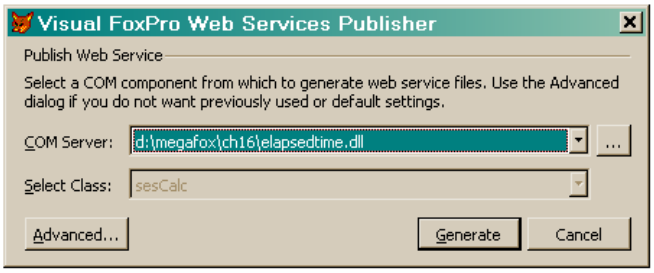


***Figure 8**. Step 3: Click on the Advanced button.*

However, hidden behind the Advanced button are a number of settings (see **Figure 9**). Visual FoxPro can, and does, supply default settings for all of these options, but you should check (if only the first time you use the wizard) that they are actually correct. The first pair of entries is concerned with the URL for, and the physical location of, the Web Services Description Language (WSDL) file that will be generated for your Web Service. (For more details about WSDL files, see the section entitled "An overview of the SOAP Toolkit" in Chapter 5.)

Notice that by default, the Web Services Publishing Wizard is set up to use an ISAPI listener. You may want to change this to specify an ASP listener because the ISAPI listener, SOAPISAP.DLL, is not automatically configured in Internet Information Server (IIS). So, if you want to use it, you must go into IIS and set it up.

Also, note that specifying an ASP listener generates an ASP page that wraps the call to the SOAP server that implements the Web Service. This page can be modified manually to add additional processing (such as parsing or verifying input parameters) before it actually calls the SOAP driver. However, if you do not need to do this sort of additional processing, the ISAPI listener is best because it delivers better performance.

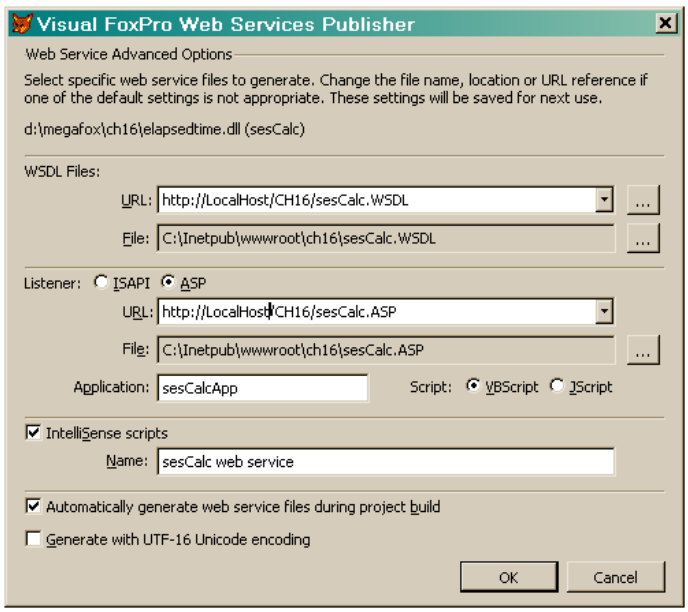


***Figure 9**. Step 4: Verify paths and default settings.*

If you want to use the IntelliSense engine to insert the code to set up the Web Service automatically, you need to check the box for "IntelliSense Scripts." The associated name that is displayed here is used to identify this Web Service to the IntelliSense engine and will appear in the list of available types whenever you use the **AS** clause of a **LOCAL** declaration on your machine.

Finally, "Automatically generate Web service files during project build" is unchecked by default. We do recommend checking this option. What it does is set up a project hook that uses the *WspHook* class from the *_WebServices* class library in the Visual FoxPro foundation classes to call the Web Service engine to rebuild the Web Service support files. It will save you a lot of work in the future because as you test your Web Service, IIS caches your COM server. To rebuild your COM server you must restart IIS; otherwise, an access denied error is generated during project build. This becomes a major pain if you must test and rebuild frequently. The Web Service project hook class has the built-in smarts to deal with this for you.

> ! *Note that the path to the WspHook class in _WebService.vcx is hard-coded into the project when this option is checked. Consequently, moving the project from one machine to another raises the error shown in **Figure 10** when you try to open it in its new location if the Visual FoxPro home directory is not the same on both machines.*

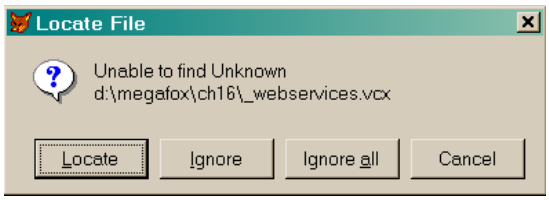


***Figure 10**. Moving the project may cause this error.*

Now you are ready to generate the files required by the Web Service. Clicking the Generate button shown in Figure 8 starts the generation process. After a few moments you should see a message similar to the one in **Figure 11**.

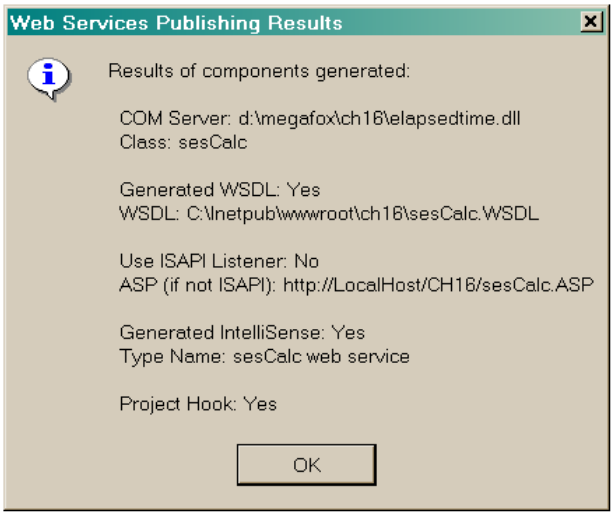


***Figure 11**. Step 5: Generate the Web Service files.*

This merely confirms that Visual FoxPro was able to generate the files it needed and that your Web service is ready for consumption.

What is a WSML file?

The Microsoft SOAP Toolkit Version 2.0, which is used to implement Web Services in Visual FoxPro 7.0, requires the presence of a Web Services Meta Language (WSML) file on the server. The WSML file provides information that maps the operations of a service (that are described in the Web Services Description Language, WSDL, file) to specific methods in the associated implementation object. It determines which object to load, and which method to call, in order to fulfill the request for a particular operation.

At the root level of a WSML file is the *ServiceMapping* element. This can have one or more *Services* and each service can have one or more *Using* and *Port* elements. The first maps a *progid* to its equivalent object, while the second defines the name used to access the object. Within the port element each of the service's methods is exposed as an *Operation*. This hierarchy is (not surprisingly) identical to the one described for the WSDL file in Chapter 5

except that it does not encompass *Parts* (which define parameters and return value). The following WSML file has a single **<service>** with one **<using>** and one **<port>** element:

```
<servicemapping name='MyComObject'>
  <service name='MyComObject'>
     <using PROGID='MyComObject.MyClass' cachable='0' ID='MyClassObject' />
     <port name='MyClassSoapPort'>
       <operation name='DoSomething'>
         <execute uses='MyClassObject' method='DoSomething' dispID='1'>
          <parameter callIndex='1' name='FirstParm'
                     elementName='FirstParm' />
          <parameter callIndex='2' name='SecondParm'
                     elementName='SecondParm' />
          <parameter callIndex='-1' name='retval'
                     elementName='Result' />
         </execute>
       </operation>
       <operation name='DoSomethingElse'>
        <execute uses='MyClassObject' method='DoSomethingElse' dispID='2'>
          <parameter callIndex='1' name='FirstParm'
                     elementName='FirstParm' />
          <parameter callIndex='2' name='SecondParm'
                     elementName='SecondParm' />
          <parameter callIndex='-1' name='retval'
                     elementName='Result' />
        </execute>
      </operation>
    </port>
  </service>
</servicemapping>
```

Structurally, the implementation of Web Services using the SOAP Toolkit V2.0 corresponds very closely to that of a standard COM DLL in that three files are also involved (see **Figure 12**).

***Figure 12**. The COM and Web Service triad of files.*

I don't expose my application on the Internet, so why should I bother with Web Services?

Although the main reason for exposing functionality as a Web Service is so that it can be accessed over the Internet using simple TCP/IP and HTTP protocols, there is another reason you may want to consider building and deploying Web Services—even for LAN-based applications. One of the main limitations of Visual FoxPro is that it does not have any back-end data processing capability; all of its work is done locally on the workstation. This can

mean that even executing a very simple query that returns only a few records can involve transferring disproportionately large amounts of data, particularly indexes, across the wire. However, a Web Service is always executed directly on the physical machine that hosts the service and returns only the results of whatever operation it performs. This means that a Web Service created with Visual FoxPro behaves just like a back-end server.

This can be a useful way of improving application performance when you need to conduct standard searches of large data tables (validating postal codes, for example). The entire workload can be localized on the server by only returning the results to the client. Such tasks are ideally suited to encapsulation as a Web Service, and the example in the next section illustrates how easily it can be done. However, before we get down to a specific example, there are a couple of issues that you have to bear in mind.

First, you do need a permanent connection to the Internet to run a Web Service. This is true even if you are running the Web Service locally (or on a file server). This is because the WSDL file automatically includes references to various standards documents that are stored on Web sites and that are used to interpret such things as data type definitions for the XML.

Second, if you need to reference data (or other files) that are external to the Web Service DLL, there are subtle, but significant, differences between the behavior of a compiled DLL as a COM component, and the same DLL exposed as part of a Web Service. Specifically we have found that Visual FoxPro functions that return paths correctly inside a DLL (or Active Server Page) do not work as expected when built into a Web Service. The best solution that we have found to date is to define the data path explicitly in our Web Service class—even when the files in question are in the same physical location as the DLL itself.

A sample Web Service (*Example: WsZip.pjx*)

The sample code for this chapter includes the project file named WSZIP.PJX that defines a project that can be compiled into a DLL designed to be exposed as a Web Service.

The Web Service accesses a Visual FoxPro table that contains a small subset of the US ZIP Code information and exposes methods to return various pieces of information from that table. The main class definition is contained in WSZIP.PRG and is based on the *ComBase* class that was described in Chapter 14.

Web Service properties

Two custom properties are explicitly defined, and populated, in WSZIP.PRG:

```
DEFINE CLASS xZipCodes AS combase OLEPUBLIC
  *** Set the name of the Error Table to Use
  cErrTable = "ZipErrors"
  *** Set the path to the Data
  cDataDir = "D:\MEGAFOX\CH16"
```

The first is used to define the name of the table that defines the error messages used by the Web Service (for an explanation of this, see the description of the *ComBase* class in Chapter 14). The second, *cDataDir*, is used to store the current physical directory to get around the problem of determining the current working directory inside a Web Service.

You may have noticed that in the *SesGenPage* class definition (GENERATEHTML.PRG), the *Init()* method included code to set the *cDataDir* property for that object. This worked well when implementing the code as a DLL or when calling it from an Active Server Page.

Unfortunately, when a DLL that contains the self-same code is exposed as a Web Service, **SYS(16)** simply does not return the path of the currently executing method. We have no idea why the function fails to return the path, but that is what happens. We cannot use the **CURDIR()** function (in either situation) because, inside the DLL, it merely returns the location of the VFP runtime library (that is, C:\WINNT\SYSTEM32 or whatever directory your particular system is set up to use).

The only way, other than hard-coding the path, that we could find was to make use of the *ServerName* property that is exposed by both the *_VFP* and the *Application* objects. The code is contained in the protected *SetUpData()* method which is called from the *Init()*.

```
************************************************************
*** [P] SETUPDATA(): Open Local copy of the ZipCode Table
************************************************************
PROTECTED FUNCTION SetUpData()
  LOCAL llOk, lcPath
  *** We need to get the path....
  IF "DLL" $ UPPER(_VFP.ServerName)
    *** Running as a Web Service
    lcPath = JUSTPATH(_VFP.ServerName )
  ELSE
    *** Running as a VFP Class
    lcPath = FULLPATH( CURDIR() )
  ENDIF
  *** Now we can open it
  WITH This
    *** Set the Directory property
    .cDataDir = lcPath
    llOK = .OpenLocalTable( "zipcodes", 3 )
    RETURN llOk
  ENDWITH
ENDFUNC
```

This does assume that the directory in which the data resides is the same as the one in which the DLL resides. However, in the case of a case of a Web Service this is not unreasonable since you must explicitly define the virtual directory anyway.

Web Service Methods

The Web Service exposes five custom methods as shown in **Table 4**.

***Table 4**. WSZip Web Service methods.*

Method	Parameters	Returns
IsZipValid	Zipcode as String	Numeric Value: • -1 for Error • 0 for Invalid Zip Code • 1 for Valid Zip Code
GetZipLine	Zipcode as String	Character String: [<City>, <State>, <zip>]
GetZipLocn	Zipcode as String	Character String: [<Latitude> <Longitude>]
GetZipAreaCode	Zipcode as String	Character String: [<Area Code> (<time zone>)]
GetZipForCity	City as String State Abbrev as String	XML String containing all matching City, State, and Zipcode values for the specified combination of City and State

In addition, the *GetErrors()* method, which is inherited from the *ComBase* class, is exposed. This returns, as always, an XML string in the following form:

```
<ERRORLOG>
  <ERRORS>
    <COUNT>000</COUNT>
    <ERRNUM>[nError]</ERRNUM>
    <ERRMSG>[cErrorText]</ERRMSG>
    <ERROCC>[cMethodName]</ERROCC>
  </ERRORS>
</ERRORLOG>
```

The actual code in the first four methods of this class is very straightforward. In each, the ZIP Code that is passed to the Web Service as a character string is validated, and used in a simple `SEEK()` to locate the appropriate record in the data table. If a record is found, the relevant result is returned.

The final method, *GetZipForCity()*, is a little more complex because it accepts either one parameter (the City name) or two parameters (City and State). If you try to call this method of the Web Service and pass only one parameter, you will get an immediate error. Interestingly, the error that is raised by Visual FoxPro is Error 1426—that is, an OLE error. Thus executing this:

```
? loSvce.GetZipForCity( "Orange" )
```

generates the error shown in **Figure 13**.

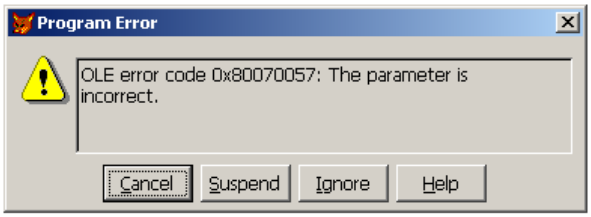


***Figure 13**. Omitting an expected parameter generates an error.*

Notice that this error is not generated by our code. The call does not even get as far as the Web Service and is actually recognized as being invalid before it ever leaves the client. It seems entirely reasonable that we should avoid incurring the overhead of a trip out to the Internet when we already know that the method call is not valid. This begs the question, how do we know the call is not valid? The answer is that the WSDL file defines how the call should be made and it is this file (that must be available to the client) that is used to validate calls to Web Service methods.

This error is avoided simply by calling the method with an empty string as the second parameter, thus:

```
? loSvce.GetZipForCity( "Orange", "" )
```

The code simply runs a query using either the name of the city, or the combination of city and state (depending up on the parameters) to retrieve matching information. This data is then packaged up as XML and returned as the response from the Web Service, like this:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<VFPData>
  <curres>
       <czipcity>Orange</czipcity>
       <czipstate>NJ</czipstate>
       <czipcode>07051</czipcode>
  </curres>
</VFPData>
```

More on Web Service behavior (*Example: CompRun.prg*)

If you create, register, and call this DLL locally, you will notice that there are significant differences in behavior when calling it as a Web Service (that is, through the interposed SOAP interfaces), compared to its behavior when you call the DLL directly (COMPRUN.PRG).

```
**********************************************************************
* Program....: COMPRUN.PRG
* Compiler...: Visual FoxPro 07.00.0000.9465
* Purpose....: Running Web Service Vs DLL
**********************************************************************

**********************************************************************
*** NOTE: This declaration is specific to how the web service was
*** registered on your machine - it may not work as posted here!!!
*** Delete this block of code, and re-create after registering
*** the web service (wszip.dll) on your own machine

*** Create the web service
LOCAL loSvce AS ZipCode Web Svce
LOCAL loWS
loWS = NEWOBJECT("Wsclient",HOME()+"ffc\_webservices.vcx")
loWS.cWSName = "ZipCode Web Svce"
loSvce = loWS.SetupClient(;
"http://ACS-SERVER/CH16/xZipCodes.WSDL", "xZipCodes", "xZipCodesSoapPort")
**********************************************************************

? "****************************"
? "*** CALL THE WEB SERVICE"
? "****************************"
? loSvce.GetZipLine( 44309  )    && "Akron, OH, 44309"
? loSvce.GetZipLine( "44309" )   && "Akron, OH, 44309"
? loSvce.GetZipLine( .F.  )      && Empty String
? loSvce.GetZipLine( DATE()  )   && Empty String
? loSvce.GetErrors()             && No errors
?

**********************************************************************
*** Now use the SAME DLL Directly
**********************************************************************
LOCAL oDL
oDL = CREATEOBJECT( 'wszip.xzipcodes' )
? "************************"
? "*** NOW CALL THE DLL"
```

```
? "************************"
? oDL.GetZipLine( 44309  )      && Empty String + Error
? oDL.GetZipLine( "44309" )     && "Akron, OH, 44309"
? oDL.GetZipLine( .F.  )        && Empty String + Error
? oDL.GetZipLine( DATE()  )     && Empty String + Error
? oDL.GetErrors()               && Three "Invalid Parameter" Errors
```

Notice that it doesn't actually matter what you pass (as the ZIP code parameter) to the Web Service. Whatever is passed gets transmitted as a string to the actual DLL, which simply sees it as an invalid code and so does not generate any error! When calling the DLL, however, the parameter is passed "as is" and so we get the "Invalid Parameter" errors.

Part of the reason for this behavior depends upon how the parameter has been declared in the source code. Thus the declaration used in WSZIP.PRG for the *GetZipLine()* method is:

```
FUNCTION GetZipLine( tcZipCode AS String ) AS String
```

When the DLL containing this method is processed to generate the WSDL files, this declaration is embedded as follows:

```
<message name='xZipCodes.GetZipLine'>
  <part name='tcZipCode' type='xsd:string'/>
</message>
```

This is required because, in order to call a Web Service, the request has to be packaged up as XML—which is, of course, actually transmitted as text and has no inherent data type. If we had omitted the **AS STRING** declaration, the WSDL file would have defined the parameter as being of data type *anyType* like this:

```
<message name='xZipCodes.GetZipLine'>
  <part name='tcZipCode' type='xsd:anyType'/>
</message>
```

which leaves the matter of deciding how to interpret the XML to the recipient.

Conclusion

Even though Visual FoxPro is not specifically designed as a Web-enabled database, that does not mean that it cannot be used in conjunction with tools that are designed specifically for use on the Web. With its support for creating and consuming XML Web Services, its ability to create COM components, and its superb string handling capabilities, Visual FoxPro can still play a significant role in supporting Web development.

Chapter 17
XML and ADO

Extending Visual FoxPro is all about communicating with other applications that are unable to use the data contained in Visual FoxPro cursors directly. This is not a problem because there are several mechanisms that we can use to make data held in Visual FoxPro available to such applications. We have covered elsewhere the use of COM components and HTML, so in this chapter we are focusing on the issues surrounding the use of more data-centric techniques using XML and ADO.

What is XML?

XML is the standard and widely used abbreviation for "eXtensible Markup Language." The roots of XML go way back to 1969 when IBM Research invented the first modern markup language, Generalized Markup Language (GML). GML was intended to be a meta-language; that is, a language that could be used to describe other languages, their grammars, and vocabularies. GML later became Standard Generalized Markup Language (SGML), which was adopted as the international data storage and exchange standard by the International Organization for Standardization (ISO) in 1986. Both XML and HTML are subsets of SGML.

The primary difference between XML and HTML is that, unlike HTML, which fuses data and presentation, XML is about data alone. Both tag semantics and the tag sets are fixed in HTML, which means sending text in HTML does not tell you anything about the data. All it tells you is how to display it in a browser. Conversely, XML lets you display and define the data in a document because it lets you define your own tags.

How does Visual FoxPro handle XML?

Two new functions, `CURSORTOXML()` and `XMLTOCURSOR()`, were introduced in Visual FoxPro 7.0 to make working with XML easier. They are, however, of limited usefulness for a couple of reasons. First, XML is hierarchical in nature but Visual FoxPro is relational. This means that representing the complex relationships inherent in Visual FoxPro data is difficult at best. `CURSORTOXML()` can be used to convert the contents of a single cursor to a single XML file. But you cannot create relational structures directly. Second, the functions rely on the XML being either entirely element-centric or entirely attribute-centric. You cannot handle elements that are qualified with attributes using these functions.

An even more serious limitation is that, when the structure of your cursor changes, so does the XML that is created using `CURSORTOXML()`. What is needed here is a layer between the cursor and the XML. To give us more control over the way in which XML is created from our cursors, we implemented a data-driven class that we use to generate XML files from our Visual FoxPro data when we need to share that data with other applications that do not understand cursors.

However, before we can talk sensibly about XML, we need to understand the terminology. So let's begin by defining some basic terms and concepts.

XML terminology

In this section we define the basic terms that you need to be familiar with in order to deal with XML.

- Attribute: An attribute-value pair, separated by an equals sign, included inside a tagged element. All values must be enclosed in quotes. Name="Marcia" is the attribute inside this tagged element <Author name="Marcia">.
- Document Element: The top-level element (root node) in an XML document that contains all the other elements. There must be one and only one document element in an XML document.
- DTD: A Document Type Definition consists of markup code that contains the grammar rules for a particular class of document. It must appear following the XML definition and prior to the document element. The syntax of the document type declaration is **<!DOCTYPE *content*>**. DTDs are old technology and not the preferred way to define the structural rules for an XML document because they do not use XML syntax. If you need to provide these rules, it is much better to go with a Schema.
- Element: An XML structural construct that consists of a starting and ending tag with information in between.
- Entity Reference: Provides ways to include information in XML documents by reference rather than by typing characters into the document directly. In other words, this is very similar to Macro substitution. This is very useful in cases where:
  - Characters can't be entered directly into a document because they would be interpreted as markup (for example, "<" and " >" symbols).
  - The content is so big that it can't be entered directly into a document because of input device limitations.
  - A document fragment appears repeatedly throughout the document.
- Namespace: A mechanism that allows developers to uniquely qualify the element names and relationships to avoid name collisions on elements that have the same name but are defined in different vocabularies. For example, if abc.dtd is a namespace that defines a name element and xyz.dtd is a namespace that defines a city element and we want to

use both of these elements in our XML document, we would first declare the namespaces like this:

xmlns:a="abc.dtd"

xmlns:x="xyz.dtd"

and we would qualify the elements with the namespace in which they are defined like this:

<a:name>Marcia Akins</name>

<x:city>Akorn</city>

- Node: Any item in an XML document. Element, attributes, and text are all examples of different types of nodes.
- Schema: A formal specification that indicates which elements are allowed in an XML document, and in what combinations. It also defines the structure of the document. It is functionally equivalent to a DTD but is written in XML. It also provides extended functionality such as data typing, so it is much more powerful than a DTD.
- Valid XML: XML that conforms to the rules defined in the XML specification, as well as the rules defined in the DTD or schema.
- Well-formed XML: XML that follows the XML tag rules listed in the W3C Recommendation for XML 1.0, but doesn't necessarily have a DTD or schema. A well-formed XML document contains one or more elements; it has a single document element, with any other elements properly nested under it. A valid document is also well-formed.
- XPath: XML Path Language is a language used by XSLT to select a set of nodes from an XML document.
- XSL: eXtensible Stylesheet Language is used to transform XML into other formats. Unlike Cascading Style Sheets, which decorates the XML tree with formatting properties, XSL transforms the tree into a new tree without altering the source document.
- XSLT: XSL transformations make use of the expression language defined by XPath for selecting elements for conditional processing and for generating text.

What parsers are available and which one should I use?

There are lots of different XML parsers available, but two of the most popular are DOM and SAX. DOM stands for Document Object Model. DOM parsers load the entire document at once and build an internal tree that can be explored hierarchically. Microsoft's XMLDOM parser is a DOM parser.

SAX stands for Simple API for XML. SAX parsers parse an XML document from top to bottom, and allow the developer to hook code into events that fire as each node is parsed. SAX is generally better suited for extremely large XML documents where it would be impractical or too slow to load the whole thing at once. The new Microsoft parser, MSXML4, has SAX interfaces that you can use in Visual FoxPro 7 because of the enhanced **DEFINE CLASS** command that includes the **IMPLEMENTS** keyword. MSXML 4.0 Service Pack 1 has a number of nice features and enhancements including a much faster parser and XSLT engine than the preceding version and support for the XML Schema language. It can be downloaded at www.microsoft.com/downloads/release.asp?releaseid=37176&area=new&ordinal=3.

Does it matter which version of MSXML I use?

The short answer here is yes. The perennial problem of creeping versionitis (a.k.a. DLL Hell) has once again reared its ugly head. The Microsoft XMLDOM is distributed in DLLs and new versions are installed in "side-by-side" mode. This has the benefit of allowing existing programs that are using older versions of the parser to continue functioning without modification. This is only a problem if you instantiate the parser using the old syntax:

```
oParser = CREATEOBJECT( 'Microsoft.XMLDOM' )
```

because this syntax uses MSXML.DLL and there are versions of that DLL (like the one that shipped with Windows 2000) that have serious bugs that make it almost unusable from Visual FoxPro. You will quickly find out if you have one of these versions by searching a document for a node that doesn't exist. The first bug rears its ugly head and the parser throws an OLE error instead of merely returning a null object, which is what should happen. Another bug is that the XMLDOM throws an "unspecified error" if you try to load invalid XML. What should happen is that the *Load()* method should return false and allow you to access to the error object. So what can you do if you have a buggy version of MSXML.DLL? Installing a newer version of the parser in "side-by-side" mode doesn't help. The simplest solution is to install a version of Internet Explorer later than V5.1, or, if you have Windows 2000, install the most recent service pack.

At the time of writing, MSXML4.DLL SP1 is the most current version of the parser. It, too, is installed in "side-by-side" mode because some obsolete and non-conformant features are no longer supported. So what does this mean? It means that if you instantiate the parser using the version-independent program ID, like this:

```
oDOM = CREATEOBJECT( 'MSXML2.DomDocument' )
```

the parser you get will not be from MSXML4.DLL. In order to take advantage of the new features in MSXML4.DLL, you must instantiate the DOM using a version-*dependent* Program ID like this:

```
oDOM = CREATEOBJECT( 'MSXML2.DomDocument.4.0' )
```

Microsoft used to provide a utility called XMLINST.EXE that could be used to run MSXML3.DLL in "Replace mode." This modified the Registry entries of earlier versions of the parser to point to MSXML3.DLL by overwriting the InprocServer32, TypeLib, and Default Icon values. This allowed legacy applications that were coded using explicit ClassIDs and ProgIDs to take advantage of the new functionality in the MSXML3.DLL without having to change any code. Unfortunately, the problems caused by running MSXML3.DLL in Replace mode were far more numerous than any benefits that it provided, so Replace mode is no longer an option with MSXML4.DLL. Another good reason for abandoning Replace mode is that the need to maintain legacy functionality bloats the component.

For similar reasons, the version independent program IDs were removed. Microsoft claims that the main reason for this was to improve code maintainability. However, they also claim that although the version independent program IDs made it easy for developers, they were error-prone and sensitive to changes in the production environment. For example, if a program is relying on MSXML3.DLL and a program that supplies its own, older, version of the DOM is installed (or even just re-installed!), the version independent ID would cause the older version to be used and this would break code. Unfortunately, using the version dependent program IDs locks us into a specific version, and when new versions with better performance and capabilities are released, our existing code cannot take advantage of them without modification.

What are the most important properties and methods of the DOM?

MSXML exposes a number of objects including:

- DomDocument, which represents the top level of the XML source.
- XMLHTTP to provide client-side protocol support for communication with HTTP servers.
- XMLSchemaCache to allow you to store multiple schema definitions and reuse them between different instance documents.
- XSLTemplate to provide support for transforming XML to HTML or XML in a different format.

In this section, we are concerned only with the DomDocument, which is the object that contains the XML that we are interested in. These, in our opinion, are the key properties:

- *Asynch*: Specifies if the XML is loaded asynchronously. The default for this property is true. The first thing you want to do after instantiating the parser is to set it to false if you want to be sure that all the XML has been loaded before processing continues.
- *Attributes*: A zero-based collection of items if the node has attributes. Only Element, Entity, and Notation nodes are allowed to have attributes. Even if one of these nodes does not have attributes, this property is not null. Instead, it is a collection that doesn't contain any items.

The attributes property of a node that is not allowed to have attributes is null. This means that oNode.Attributes.Length = 0.

- *ChildNodes*: A node list object that contains all the children of this node. Like Attributes, if the current node has no children, its *ChildNodes* property is a collection that doesn't contain any items. This means that oNode.ChildNodes.Length = 0.
- *DocumentElement:* Contains the root node of the document.
- *Length*: Returns the number of items in a collection.
- *NextSibling*: Contains an object reference to the node at the same level of hierarchy that follows the current node.
- *NodeName*: The qualified name of the current node.
- *NodeType*: A numeric constant that specifies the XML Document Object Model (DOM) node type. This determines valid values and whether the node can have child nodes. Node types include, but are not limited to, the following:
  - 1: Element
  - 2: Attribute
  - 3: Text
  - 4: CDATA Section
- *NodeTypeString*: The type of node, expressed as a character sting, instead of one of the *NodeType* constants defined previously.
- *NodeValue*: The text associated with a node.
- *ParentNode*: The parent of the current node. Note that if the current node has been created, but has not yet been added to the tree, its *ParentNode* is NULL. All nodes, except for the Document node, DocumentFragment, and Attribute can have a parent.
- *PreviousSibling*: The node before the current node at the same level of hierarchy.
- *TagName*: The element name for element nodes. Similar to *NodeName*.
- *Text*: The concatenated text of a node that includes all of its descendents. To find the text associated with individual elements, you must use the *NodeValue* of the text node associated with that element.
- *Value*: Property of an attribute node that contains its value.
- *XML*: Contains the XML representation of a node and all of its descendents.

One thing to be aware of is that most of the properties are case-sensitive. Tag names, attribute names, and attribute values all enforce case-sensitivity when you refer to items in the document that you are parsing.

The DomDocument interface exposes numerous methods that can be used to manipulate the XML document. As you can see from the following list, the DOM has several methods that allow you to accomplish the same task. (For a complete list, refer to the XML SDK.) Some of these methods, like *AppendChild()*, are used to create or modify a document. Others, like *GetAttribute()*, are used to parse a document.

- *AppendChild*: Adds the specified node as the last child of the parent node. This method takes an object reference to the child node. The child node can be a new node (created using the *CreateNode()* method) or an existing node. If the child node has an existing parent in the tree, it is removed from that parent and inserted in the new location.
- *CreateAttribute*: Creates an attribute with the specified name. Although this method creates a new object in the context of the document, it is not associated with any element in the tree until the parent element's *SetAttributeNode()* method is invoked. Even after the attribute is associated with an element, its *ParentNode* property remains NULL because, while the element "owns" the attribute, it is not, in this context, its "parent."
- *CreateElement*: Creates an element node with the specified *NodeName*. Like *CreateAttribute* (and all the rest of the *Create…()* methods), this method creates a new object but does not add it to the tree. You must invoke the *AppendChild()*, *InsertBefore()*, or *ReplaceChild()* of an existing node to do this.
- *CreateNode*: Creates a node of the specified type. You cannot use this method to create Document, DocumentType, Entity, or Notation nodes.
- *GetAttribute*: Method of an element node that, when passed the *NodeName* of one of the element's attribute nodes, returns its *Value*.
- *GetAttributeNode*: Method of an element node that, when passed the *NodeName* of one of its attribute nodes, returns an object reference to that node (or NULL if it does not exist).
- *GetElementsByTagName*: Passed the Tag name of an element, return a *NodeList* object that contains all the nodes in the document that have that *NodeName*.
- *GetNamedItem*: A method of the *Attributes* collection of an element node that returns an object reference to the attribute with the specified name.
- *HasChildNodes*: Returns true if the passed node has children.
- *Item*: Allows random access to individual nodes within a collection. You can use this syntax: loNodeList.Item(lnCnt) to iterate through the

nodes in the list. A node list object is returned from methods such as *GetElementsByTagName()* and *SelectSingleNode()*. The *ChildNodes* property of an element node.

- *Load*: Loads an XML document from the specified file.
- *LoadXML*: Loads an XML document from the specified XML string.
- *NextNode*: Returns an object reference to the next node in the collection and is an alternative to using the *Item* collection to iterate through a collection of nodes. This code:

```
For lnCnt = 1 TO loNodeList.Length
  ?loNodeList.Item( lnCnt - 1 ).NodeName
EndFor
```

does exactly the same thing as this:

```
loNode = loNodeList.NextNode
DO WHILE VARTYPE( loNode ) = 'O'
  ?loNode.NodeName
  loNode = loNodeList.NextNode()
ENDDO
```

- *Save*: Saves an XML document to the specified location
- *SelectNodes*: Passed a string that specifies a pattern-matching operation to be applied, returns a *NodeList* object containing all matching nodes. If the DOM's *SelectionLanguage* property has been set to XPath, this string is an XPath expression. Otherwise, it is an XSL patterns query.
- *SelectSingleNode*: Returns an object reference to the first node that matches the specified pattern.
- *SetAttribute*: Method of an element node used to add an attribute. When passed the name of an attribute and a value, sets the value of the named attribute.

How do I data-drive the production of XML? (*Example: GenerateXml.scx and ExportXML.prg::ExportXML*)

As noted earlier, one of the limitations to using the **XMLTOCURSOR()** function is that it cannot be used to represent complex relationships between multiple cursors in a single document. Another problem is that when the structure of a cursor changes, so does the generated XML. Fortunately, since we are using Visual FoxPro, it is possible to create a data-driven class that gives us the flexibility we require. However, you should be aware that our sample classes do not account for every single type of node that could be contained in an XML document, nor do

they handle DTDs or Schemas. They are intended to show how an XML handling class can be constructed to meet your specific needs.

We designed our metadata to handle the creation of XML from specified cursors as well as the creation of these cursors from an XML document. This metadata is contained in two separate tables. The first one, XMLCURSORS.DBF, holds information about the cursors from which the XML is generated or into which XML is to be imported. Its structure is shown in **Table 1**.

***Table 1**. Structure of XMLCURSORS.DBF.*

Field name	Explanation
cProcName	A descriptive name that ties together all of the records used to generate a single XML document.
cAlias	The name of the cursor to use.
cTag	The name of the controlling index tag.
cPkField	The name of the primary key field.
cFkField	The name of the foreign key field (if applicable).
iLevel	The level of hierarchy that this cursor occupies. A level 1 cursor has no cFKField specified.
cFkType	Data type of the foreign key field (should be integer, but the Tastrade sample data uses character PK fields).
iFkLen	Length of the foreign key field.

The second table, XMLMAP.DBF, defines the nodes in the XML document and their relationships to each other. It also specifies the rules for populating the text nodes from the cursors (see **Table 2**). Note that the structure of this table does not differentiate between elements and attributes. Conceptually, it treats attributes as children of the element that they qualify.

***Table 2**. Structure of XMLMAP.DBF.*

Field name	Explanation
cProcName	A descriptive name that ties together all of the records used to generate a single XML document.
cNodeName	The text to use as the element tag or, in the case of an attribute, the attribute name.
cNodeType	For this demonstration, ELEMENT or ATTRIBUTE.
mNodeText	Rule to use to create the Text node for a given element or attribute.
cParent	The *NodeName* (as specified by the *cNodeName* field) of the parent node.
lSeq	The sequence number for this node with respect to its parent.
cAlias	The alias into which this node's information is imported / from which it is exported.
cField	The field in the target alias.
cType	Data type of the field in the import cursor.
iLen	Field length of the field in the import cursor.
iDecimals	Number of decimal places for the field in the import cursor.
lJustify	When true, the field is a right justified character field (to accommodate to goofy right justified PKs in the Tastrade sample data).

Very little code is required to generate the document using the DOM. This is because it automagically "knows" where to put the closing tags and how to embed attributes inside the elements that own them. However, you incur a performance penalty for the convenience of using the DOM. If performance is critical, you can use Visual FoxPro's great string handling capabilities to create the document. However, if you use Visual FoxPro to generate the document, you must write code to determine where you are in the document and insert the closing tags where required. Using Visual FoxPro to generate the document without the aid of the DOM also requires that you write code to manually embed attributes within the opening tag of their owning element. Since part of our goal here is to illustrate how the DOM works, our class uses the DOM to generate the document.

The *ExportXML* class has six custom properties:

- *cVersion*: The version-dependent ProgID of the parser to instantiate. This defaults to MSXML2.DomDocument.4.0. If you do not have MSXML4.DLL installed, you must change this before running the sample.
- *cProcName*: The descriptive name that ties together all of the records in the metadata used to generate the XML. This is assigned at runtime by the calling process.
- *oXML*: Object reference to the DOM after it is instantiated.
- *cCondition*: Condition used to limit the number of records processed for inclusion in the XML document.
- *cXMLFile*: Name of the file into which the XML document should be saved.
- *aCursors*: Array property that holds information about the cursors used to generate the XML document.

The sample form (see **Figure 1**) uses the data from the Tasmanian Traders sample application that ships with Visual FoxPro. It allows you to select a customer before clicking the Generate XML button to create an XML file consisting of the customer and order information. In order to generate the XML file for the selected customer, the form creates an instance of the *ExportXML* class when it is instantiated. The form's custom *GenerateXML()* method is invoked from the *Click()* method of the Generate XML button. This form method sets the *cProcName* and *cCondition* properties of the *ExportXML* object before calling the exporter's custom *GenerateXML()* method.

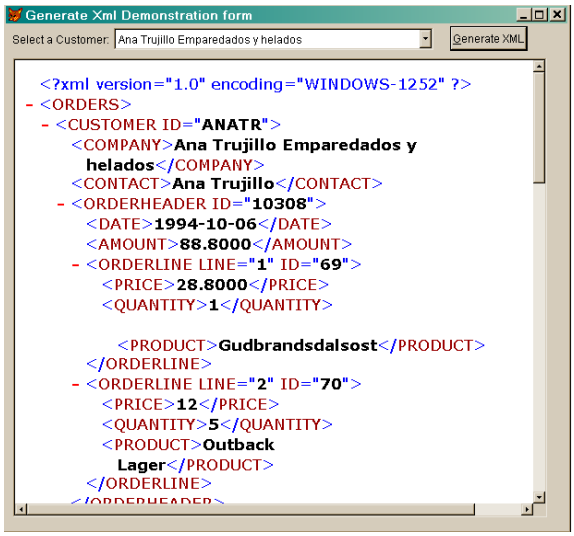


***Figure 1**. Data-driven production of XML from Visual FoxPro cursors.*

Assigning the *cProcName* to the XML exporter object fires off an assign method that sets the class up to generate the specified XML. After saving the assigned *cProcName* as uppercase, this method populates the *aCursors* array from XMLCURSORS.DBF and ensures that the mapping table, XMLMAP.DBF, is open and its order set.

```
This.cProcName = UPPER( ALLTRIM( tcProcName ) )
SELECT cAlias, cPKField, cFkField, cTag, iLevel ;
  FROM XmlCursors WHERE UPPER( ALLTRIM( cProcName ) ) == This.cProcName ;
  ORDER BY iLevel INTO ARRAY This.aCursors
IF NOT USED( 'XMLMap' )
  USE XMLMap IN 0
  SELECT XMLMap
  SET ORDER TO cProcName
ENDIF
```

Next, the assign method loops through the *aCursors* array ensuring that the tables required to generate the XML are open.

```
lnLen = ALEN( This.aCursors, 1 )
FOR lnCnt = 1 TO lnLen
  lcAlias = ALLTRIM( This.aCursors[ lnCnt, 1 ] )
  lcTag = ALLTRIM( This.aCursors[ lnCnt, 4 ] )
  IF NOT USED( lcAlias )
    USE ( lcAlias ) AGAIN IN 0
  ENDIF
  SELECT ( lcAlias )
  IF NOT EMPTY( lcTag )
    SET ORDER TO ( lcTag )
  ENDIF
ENDFOR
```

Finally, The DOM is instantiated and set up for synchronous operation.

```
  This.oXML = CreateObject( This.cVersion )
  This.oXML.async = .F.
```

The custom *GenerateXML()* method controls the creation of the XML document. It locates the record in XMLMAP.DBF that is used to generate the *DocumentElement*. This record does not have anything specified in its *cParent* field. In our example, this is the record where the *cNodeName* field contains the value *ORDERS*.

After adding the *DocumentElement* to the document, *GenerateXML()* goes on to process the cursor that is defined as the top level of the hierarchy; that is, the cursor referenced by the first row of the custom *aCursors* array property. This cursor is used to generate all of the *ChildNodes* of the *DocumentElement*. In the case of our example, this is CUSTOMER.DBF.

```
*** Create the root element
*** This is represented in the metadata by the
*** record for the process that has no parent record
SELECT XMLMap
LOCATE FOR UPPER( ALLTRIM( XMLMap.cProcName ) ) == This.cProcName ;
       AND EMPTY( XMLMap.cParent )
IF NOT FOUND()
  ASSERT .F. MESSAGE 'Unable to find XML information in metadata'
  RETURN .F.
ENDIF

WITH This.oXML
  loRoot = .CreateElement( ALLTRIM( XMLMap.cNodeName ) )

  *** Now add the root element to the document
  .AppendChild( loRoot )

  *** Process the first level
  lcAlias = UPPER( ALLTRIM( This.aCursors[ 1, 1 ] ) )
  lcpkField = UPPER( ALLTRIM( This.aCursors[ 1, 2 ] ) )
  SELECT ( lcAlias )
```

The *GenerateXML()* method invokes the exporter's custom *AddChildren()* method for each record that it processes. In the case of our example, only a single record is processed: the customer record that we chose in the dropdown list of the example form. However, this class can also handle the case where the *DocumentElement* has multiple children.

```
  *** See if we are processing for some condition
  IF NOT EMPTY( This.cCondition )
    SCAN FOR EVAL( This.cCondition )
      *** Add the child nodes for this level
      SELECT * FROM XMLMap WHERE ;
        UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
        UPPER( ALLTRIM( cAlias ) ) = lcAlias ;
        ORDER BY cParent, iSeq INTO CURSOR csrKids NOFILTER

      **** create the nodes in the document
      **** according to the metadata
      loNode = This.AddChildren( loRoot )
```

The next step is to invoke the custom *ProcessCursors()* method to process the cursors at the lower levels of the hierarchy.

```
      IF ALEN( This.aCursors, 1 ) > 1
        This.ProcessCursors( 2, loNode )
      ENDIF
    ENDSCAN
  ENDIF
```

Finally, after the entire document tree is constructed, the *GenerateXML()* method adds an encoding declaration and writes the document out to a file. We explicitly add the encoding declaration because when no encoding attribute is specified, the default setting is UTF-8. This is necessary because the customer table that ships with the Visual FoxPro sample application contains characters (such as è and ó) that cannot be interpreted correctly using UTF-8 encoding. Without this declaration, the resulting XML file cannot be viewed in Internet Explorer because of parser errors.

```
  *** Now save the XML as a file
  lcStr = '<?xml version="1.0" encoding="WINDOWS-1252"?>' + .XML
  STRTOFILE( lcStr, This.cXMLFile )

  *** Finished...so release the parser
  This.oXML = .NULL.
ENDWITH
```

The custom *AddChildren()* method adds nodes to the document for the specified *ParentNode*. It expects to be passed an object reference to the required *ParentNode*. It then iterates through all the records in the metadata where the *cAlias* field matches the name of the cursor currently being processed. Child nodes are created according to whatever rules are defined in the metadata. Because this method is called recursively, the first thing that it must do is save the current position in the metadata so that it can be restored upon returning from each call. Next, it scans the metadata for all the records that have a *cParent* field that matches the *NodeName* of the node that it was passed.

```
lnRecNo = RECNO( 'csrKids' )

*** See if this node has kids
SELECT csrKids
LOCATE FOR UPPER( ALLTRIM( csrKids.cParent ) ) == ;
       UPPER( ALLTRIM( toParent.NodeName ) )
IF FOUND()
  SCAN WHILE UPPER( ALLTRIM( csrKids.cParent ) ) == ;
       UPPER( ALLTRIM( toParent.NodeName ) )
```

The only *NodeTypes* that our class handles are Elements and Attributes. So the next bit of code checks to see what type of node is specified in the metadata and creates it.

```
    IF UPPER( ALLTRIM( csrKids.cNodeType ) ) == 'ATTRIBUTE'
      loNode = This.oXML.CreateAttribute( UPPER( ALLTRIM( csrKids.cNodeName )))
    ELSE
      loNode = This.oXML.CreateElement( UPPER( ALLTRIM( csrKids.cNodeName )))
    ENDIF
```

Next, the *AddChildren()* method processes the *mNodeText* field of the metadata. This field specifies an expression that, when evaluated, produces any text that is associated with the element or attribute that was just created. After this text is obtained, it must be cleansed of any characters that the XML parser would misinterpret as markup. Finally, a text node is created to hold the text and is associated with its *ParentNode.*

```
    IF NOT EMPTY( csrKids.mNodeText )
      lcText = EVALUATE( ALLTRIM( csrKids.mNodeText ) )
      lcText = STRTRAN( STRTRAN( STRTRAN( lcText, "&", '&amp;'),;
                        '"', '&quot;' ), "'", '&apos;' )
      lcText = STRTRAN( STRTRAN( lcText, "<", '&lt;'), '>', '&gt;' )
      loNodeText = This.oXML.CreateTextNode( lcText )
      *** And append the text node to its parent node
      loNode.AppendChild( loNodeText )
    ENDIF
```

At this point, we have either a newly created *Attribute* or *Element* node and its associated *TextNode*. However, this node is not yet part of the XML document. It must be explicitly added to the document like this:

```
IF UPPER( ALLTRIM( csrKids.cNodeType ) ) == 'ATTRIBUTE'
  toParent.SetAttributeNode( loNode )
ELSE
  toParent.AppendChild( loNode )
ENDIF
```

The *AddChildren()* method finishes by calling itself recursively so that it can add any children that are specified for the newly created node in the metadata. When it finally returns to the original caller, it passes back an object reference to the first child node that was added.

The custom *ProcessCursors()* method begins by processing the cursor specified in the second row of the *aCursors* array. It processes each record in the cursor and calls *AddChildren()* to add the required nodes for each of them. Only records that are related to the parent cursor (the cursor referenced in the previous row of the *aCursors* array) are included for processing. The method then calls itself recursively until all the cursors have been processed.

```
*** Get the name of the parent alias and the pk field
lcAlias = UPPER( ALLTRIM( This.aCursors[ tiLevel - 1, 1 ] ) )
lcpkField = UPPER( ALLTRIM( This.aCursors[ tiLevel - 1, 2 ] ) )

*** And the child info
lcChildAlias = UPPER( ALLTRIM( This.aCursors[ tiLevel, 1 ] ) )
lcFkField = UPPER( ALLTRIM( This.aCursors[ tiLevel, 3 ] ) )
luValue = EVALUATE( lcAlias + '.' + lcPkField )

*** And start adding the child nodes
SELECT ( lcChildAlias )
SCAN FOR EVALUATE( lcChildAlias + '.' + lcFkField ) == luValue
  SELECT * FROM XMLMap WHERE ;
      UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
      UPPER( ALLTRIM( cAlias ) ) == lcChildAlias ;
      ORDER BY cParent, iSeq INTO CURSOR csrKids NOFILTER
  **** create the nodes in the document
```

```
  **** according to the metadata
  loNode = This.AddChildren( toNode )

  *** And see if this node has children
  IF tiLevel < ALEN( This.aCursors, 1 )
    This.ProcessCursors( tiLevel + 1, loNode )
  ENDIF
ENDSCAN
```

How do I data drive importing XML into cursors?

We can use the same metadata that we used in the previous section to convert XML that is nested hierarchically into multiple cursors. However, in this case we can choose between using the DOM and using the SAX interface to accomplish the task. Using the SAX interface is much faster for large documents because it does not have to read the entire document into memory before it begins processing. However, it lacks the power of the DOM to search for specific nodes in the tree structure. If you are parsing small documents, the SAX interface may actually be slower than the DOM because of the extra objects and interfaces that are required.

How do I use the SAX interface to import XML? (*Example: SAXImport.scx and SAXHandler.prg::VFPSAXHandler*)

In order to use the SAX interfaces, you must be running Visual FoxPro 7 because previous versions do not allow us to implement interfaces. For an in-depth discussion of interface implementation, refer to Chapter 14, "VFP and COM." The easiest way to begin creating a class that implements the SAX interface is to open the program file that contains the class definition and to open MSXML4.DLL in the Object Browser. We can then easily drag the interfaces that we need from the Object Browser into our program file, and the methods defined in that interface are created for us automatically. All that remains is to write the required code in the appropriate methods.

One of the confusing things about the defined interfaces for MSXML4.DLL is that there seem to be duplicates. For example, there is an interface called *ISAXContentHandler* and there is another one called *IVBSAXContentHandler*. The reason for this is that SAX was originally defined for the Java programming language using Java interface definitions, and these interfaces are not language-neutral. When Microsoft added support for SAX in MSXML 3.0, it included support for both C++ and Visual Basic. Each of these language bindings requires a different set of interfaces that reflect the individual language and type restrictions. So interfaces that are prefixed with "ISAX" are the interfaces for C++ and those that begin with "IVBSAX" are meant to be used in Visual Basic. When we implement these interfaces in Visual FoxPro, we need to use the interfaces that are supported in VB.

The SAX parser does not treat the XML document as a tree structure. Instead, it parses the document from top to bottom, and as it does various events are fired. We can write method code to do specific processing when these events occur. For example, the *StartElement* event of the *ContentHandler* interface receives notification of the beginning of an element. So we can write method code here to do anything that needs to be done to process elements.

The sample form (see **Figure 2**) uses the SAX interfaces and a data-driven approach to import customer orders from an XML document into three cursors.

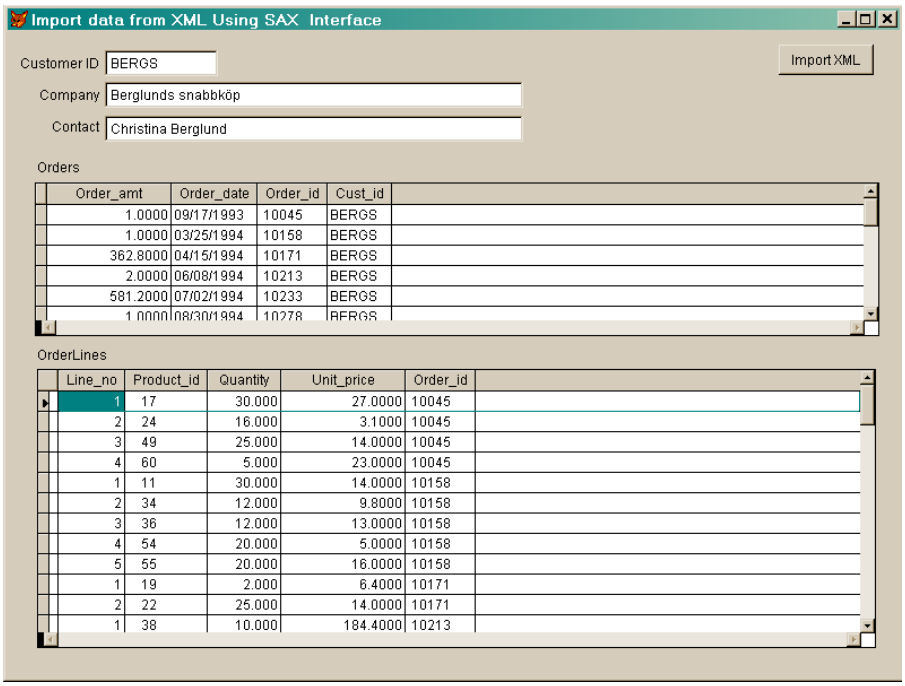


***Figure 2**. Data-driven import of XML into Visual FoxPro cursors using SAX interface.*

When the Import XML button is clicked on the sample form, it invokes the form's custom *ImportXML()* method, passing it the name of the XML file to import. The method begins by creating an instance of the SAX Reader object.

```
loReader  = CREATEOBJECT("MSXML2.SAXXMLReader.4.0")
```

The SAX Reader allows an application to register event handling for document processing and to initiate the parsing process. After it is instantiated, a *ContentHandler* and *ErrorHandler* must be instantiated and set as properties of the Reader object.

```
Thisform.oHandler = CREATEOBJECT("VFPSAXhandler")
loReader.contentHandler = Thisform.oHandler
loReader.ErrorHandler = Thisform.oHandler
```

Next, the handler's *cProcName* property is set and the *Parse()* method of the SAX reader is invoked, passing it the XML string to process.

```
Thisform.ohandler.cProcName = 'XMLORDERS'
loReader.Parse( FILETOSTR( tcFileName ) )
```

Finally, the form refreshes its controls to display the results of the import process.

The *VFPSAXHandler* class implements the *IVBSAXContentHandler*, *IVBSAXDTDHandler*, and *IVBSAXErrorHandler* interfaces in MSXML4.DLL. It has the following custom properties:

- *cProcName*: The descriptive name that ties together all of the records in the metadata used to generate the cursors from XML. This is assigned at runtime by the calling process.
- *cParent*: The name of the parent of the node being processed.
- *cNodeName*: The name of the node being processed.
- *cNodeText*: The value of the text node being processed.
- *cAlias*: The name of the cursor currently being populated.
- *aCursors*: The array of cursor names to be populated.
- *aParents*: The array of all the nodes in XMLMAP.DBF that have children.

When the calling process sets the object's *cProcName* property, it fires off an assign method that begins by populating the handler's two custom array properties from the metadata.

```
SELECT cAlias, cPKField, cFkField, cTag, iLevel ;
  FROM XMLCursors WHERE UPPER( ALLTRIM( cProcName ) ) == This.cProcName ;
  ORDER BY iLevel INTO ARRAY This.aCursors
*** And get the names of all the parent nodes
SELECT DISTINCT UPPER( cParent ) AS cParent ;
  FROM XMLMap INTO ARRAY This.aParents
```

Next, the assign method creates each of the cursors that are required to store the information in the XML document. It is likely that the information that is imported will require further processing by the system. For example, the system may need to decide whether a given record is an insert or an update and take appropriate action. For this reason, the cursors created by the *VFPSAXHandler* use the alias names in the metadata prefixed with "cur". Once the data has been imported into these cursors, the calling process can go on to do any instance specific processing.

```
lnLen = ALEN( This.aCursors, 1 )
FOR lnCnt = 1 TO lnLen
  lcAlias = UPPER( ALLTRIM( This.aCursors[ lnCnt, 1 ] ) )
  SELECT DISTINCT cField, cType, iLen, iDecimals FROM XMLMap ;
    WHERE UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
          UPPER( ALLTRIM( cAlias ) ) ==  lcAlias AND ;
          NOT EMPTY( cField ) INTO ARRAY laStru

  *** Now see if we need to add a foreign key field to the array
  SELECT XMLCursors
  LOCATE FOR UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
    UPPER( ALLTRIM( cAlias ) ) == lcAlias
  IF FOUND() AND NOT EMPTY( XMLCursors.cFkField )

    *** make sure it is not already in the array
    IF ASCAN( laStru, ALLTRIM( XMLCursors.cFkField ), -1, -1, 1, 15 ) = 0
```

```
      *** Add the foreign key to the array
      lnArrayLen = ALEN( laStru, 1 ) + 1
      DIMENSION laStru[ lnArrayLen, 4 ]
      laStru[ lnArrayLen, 1 ] = ALLTRIM( XMLCursors.cFkField )
      laStru[ lnArrayLen, 2 ] = ALLTRIM( XMLCursors.cFkType )
      laStru[ lnArrayLen, 3 ] = XMLCursors.iFkLen
      laStru[ lnArrayLen, 4 ] = 0
    ENDIF
  ENDIF
  CREATE CURSOR ( 'cur' + lcAlias ) FROM ARRAY laStru
ENDFOR
```

When the form calls the SAX Reader's *Parse()* method, the entire document is parsed from beginning to end. As it is parsed, various events of the content handler are fired. This code, in the *StartElement()* method, executes whenever an element is encountered in the document. One of the parameters passed to this method is the name of the element currently being processed, and this parameter is saved to the *cNodeName* property. The method then ensures that its custom *cParent* property contains the correct information and positions XMLMAP.DBF on the record that contains the processing information for the current element.

```
This.cNodeName = UPPER( ALLTRIM( strLocalName ) )

*** See if we need to reset the current parent
SELECT * from XMLMap WHERE ;
  UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
  UPPER( ALLTRIM( cNodeName ) ) == This.cNodeName INTO CURSOR Junk

*** If we have a unique record, get the parent from it
IF _TALLY = 1
  This.cParent = UPPER( ALLTRIM( Junk.cParent ) )
ENDIF

*** Now find the correct record in the mapping table
SELECT XMLMap
LOCATE FOR UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
  UPPER( ALLTRIM( cNodeName ) ) == This.cNodeName AND ;
  UPPER( ALLTRIM( cParent ) ) == This.cParent
IF FOUND()
  *** See if this node is a parent node and save it as the current parent
  lnRow = ASCAN( This.aParents, This.cNodeName, -1, -1, 1, 15 )
  IF lnRow > 0
    This.cParent = UPPER( ALLTRIM( This.aParents[ lnRow ] ) )
  ENDIF
```

The next thing that is required is to see whether a record must be inserted into the cursor that is currently being processed. Whenever the alias being processed does not match the alias in the current record in the metadata, a new record must be inserted.

```
  IF NOT EMPTY( XMLMap.cAlias )
    IF UPPER( ALLTRIM( XMLMap.cAlias ) ) == This.cAlias
      *** No need to add a record
    ELSE
      This.cAlias = UPPER( ALLTRIM( XMLMap.cAlias ) )
      APPEND BLANK IN ( 'cur' + This.cAlias )
```

```
      *** And see if we need to populate a foreign key field
      lnRow = ASCAN( This.aCursors, This.cAlias, -1, -1, 1, 15 )
      IF lnRow > 0
        *** Get the name of the field
        lcFkField = This.aCursors[ lnRow, 3 ]
        IF NOT EMPTY( lcFkField )

          *** And get the value of the pk from the parent table
          luValue = EVALUATE( 'cur' + This.aCursors[ lnRow - 1, 1 ] + ;
                    '.' + This.aCursors[ lnRow - 1, 2 ] )
          REPLACE ( lcFkField ) WITH luValue IN ( 'cur' + This.cAlias )
        ENDIF
      ENDIF
    ENDIF
```

Now we must check to see whether the current node has any attributes. An object containing the attributes collection of the current element is passed to the *StartElement()* method as one of its parameters. All that needs to be done is to iterate through the collection to extract the name and value of each attribute. We can use the name of the attribute and the name of its parent element to locate the correct processing record in the metadata. Finally, since all data in an XML document is character data, we must convert the attribute's value to the data type of the field we need to update. This is accomplished using the *VFPSAXHandler*'s custom *Str2Exp()* method.

```
FOR lnCnt = 0 TO oAttributes.Length -1
  lcAttribute = UPPER( ALLTRIM( oAttributes.getQName( lnCnt ) ) )
  lcValue = oAttributes.getValue( lnCnt )
  *** Now see where we need to plug the value in
  SELECT XMLMap
  LOCATE FOR UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
    UPPER( ALLTRIM( cParent ) ) == This.cParent AND ;
    UPPER( ALLTRIM( cNodeName ) ) == lcAttribute AND ;
    UPPER( ALLTRIM( cNodeType ) ) == "ATTRIBUTE"
  IF FOUND() AND NOT EMPTY( XMLMap.cField )
    *** Go ahead and plug in the value
    REPLACE ( ALLTRIM(XMLMap.cField ) ) WITH ;
      This.Str2Exp( lcValue, XMLMap.cType, XMLMap.iLen, XMLMap.lJustify ) ;
      IN ( 'cur' + This.cAlias )
  ENDIF
ENDFOR
```

The *Characters* event is fired whenever a text node is encountered by the SAX parser and passed the node's *Text* as a parameter. However, we cannot just take the passed parameter and use it to update the cursor because of the way entity references are handled. For example, if the *NodeText* contains "B's Beverages", the *Characters* event is fired three times in succession. The first time it is fired, it passes the value "B". The second time it passes "&". And the third time it passes "apos;s Beverages". So in order to capture the value correctly, we must save the value of the parameter to the object's custom *cNodeText* property and concatenate successive values like this:

```
IF NOT EMPTY( strChars )
  This.cNodeText = This.cNodeText + strChars
ENDIF
```

The *EndElement* event is fired when the closing tag of an element is encountered. Very little code is required in this method to update the correct cursor. The handler's custom *cAlias* property contains the name of the alias that is currently being processed, and its *cNodeName* property holds the name of the parent element. So all we need to do is locate the correct record in XMLMAP.DBF and apply the rules to update the appropriate field with the value that we were passed before resetting the custom *cNodeTest* property to an empty string.

```
IF NOT EMPTY( This.cNodeText )

  *** translate any named entities like apostrophes and quotes
  tcValue = STRTRAN( STRTRAN( This.cNodeText, '&lt;', "<" ), '&gt;', '>' )
  tcValue = STRTRAN( STRTRAN( STRTRAN( tcValue, '&quot;', '"' ), ;
                    '&apos;', "'" ), '&amp;', "&" )
  SELECT XMLMap
  LOCATE FOR UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
    UPPER( ALLTRIM( cAlias ) ) == This.cAlias AND ;
    UPPER( ALLTRIM( cNodeName ) ) == This.cNodeName AND ;
    UPPER( ALLTRIM( cNodeType ) ) == "ELEMENT"
  IF FOUND() AND NOT EMPTY(XMLMap.cField )
    REPLACE ( ALLTRIM(XMLMap.cField ) ) WITH ;
      This.Str2Exp( tcValue, XMLMap.cType, XMLMap.iLen ) ;
      IN ( 'cur' + This.cAlias )
  ENDIF
ENDIF
This.cNodeText = ""
```

How do I use the DOM to import XML? (*Example: DOMImport.scx and DomHandler.prg::VFPDOMHandler*)

Our *VFPDOMHandler* uses the same data-driven methodology as the *VFPSAXHandler* described earlier. The sample form's *ImportXML()* method has functionality that is very similar to that of SAXIMPORT.SCX. It creates an instance of the *DOMHandler* and points the *VFPDOMHandler* to the form's *DataSession* before setting its *cProcName* and invoking its custom *Parse()* method. The *VFPDOMHandler*'s *DataSessionId* must be set to that of the form because the class is based on the session class. This way the *VFPDOMHandler* can have its own private data session when compiled into a DLL. This, of course, poses a small technical problem when instantiating the object in a form. If we do not set it up to share the form's *DataSession*, the form is unable to see any of the cursors created from the XML document.

When the *VFPDOMHandler*'s *Parse()* method is invoked, it loads the XML file and passes the *DocumentElement* to its custom *ProcessNode()* method.

```
WITH This.oXML
  .load( tcFile )
  This.ProcessNode( .DocumentElement )
ENDWITH
```

ProcessNode() controls the processing of the document. The processing sequence is:

1. Update the custom *cParent* property with *NodeName* of the current node's parent.

2. Locate the record for this element in XMLMAP.DBF.
3. If the current node is an element, determine whether the current alias has changed and add a record to the newly selected alias when it does.
4. Process any attributes that belong to the current element.
5. If the current node is a text node, use its *NodeValue* to update the appropriate field in the current alias.
6. Process any child codes of the current node.

```
*** Find the correct record in the mapping table for this node
*** If this is the root node, we must set its parent to an empty string
*** otherwise, the parent is the NodeName of the parent node
IF toNode.ParentNode.NodeType = NODE_DOCUMENT
  lcParent = ''
ELSE
  lcParent = UPPER( toNode.ParentNode.NodeName )
ENDIF

*** Now see if we have switched cursors because if we have,
*** we need to insert a new record into a cursor
IF toNode.NodeType = NODE_ELEMENT
  SELECT XMLMap
  LOCATE FOR UPPER( ALLTRIM( cProcName ) ) == This.cProcName AND ;
    UPPER( ALLTRIM( cNodeName ) ) == UPPER( ALLTRIM( toNode.NodeName ) ) AND ;
    UPPER( ALLTRIM( cParent ) ) == lcParent
  IF FOUND()

    *** If we have an associated cursor, see if we need to add a record
    IF NOT EMPTY(XMLMap.cAlias )
      IF UPPER( ALLTRIM(XMLMap.cAlias ) ) == This.cAlias
        *** No need to add a record
      ELSE
        This.AddNewRecord()
      ENDIF
    ENDIF
  ENDIF
ENDIF

*** Next see if we have any attributes
This.GetAttributes( toNode )

*** if this is a text node, use it to update the correct field
IF toNode.NodeType = NODE_TEXT
  This.UpdateField( toNode.NodeValue, toNode.ParentNode.NodeName, ;
    toNode.ParentNode.ParentNode.NodeName )
ENDIF

*** See if this node has children to process
IF toNode.HasChildNodes
  FOR EACH lochild in toNode.ChildNodes
    This.ProcessNode( loChild )
  ENDFOR
ENDIF
```

The code in the custom *AddNewRecord()* and *UpdateField()* methods is very similar to the code we included in the discussion of our *VFPSAXHandler* class in the previous section. The custom *GetAttributes()* method merely extracts any attributes owned by the current element and uses them to update the appropriate fields.

How do I validate an XML document using a schema?

There are times when it is critical that an XML document contain valid data, and in these circumstances a schema can be used to validate its content. Consider the perennial problem that occurs whenever we have to import data into our system from an external source. Occasionally, the sender alters the structure of the file they are sending and does not notify us of the changes. Code breaks and we spend lots of wasted hours trying to track down the problem only to find that it is the structure of the import file. If we use a schema to validate incoming XML, we know immediately when there is a problem with its structure or content.

There are two different types of schemas from which you can choose: XSD and XDR. The XML Schema definition language (XSD) is the current World Wide Web Consortium (W3C) specification for XML schemas. XML-Data Reduced (XDR) refers to the subset of the XML-Data schema specification that was implemented by Microsoft before the existence of XSD as a recommended standard by the W3C.

How do I create an XDR schema? (*Example: OrderSchema.xml*)

When defining a schema for a particular class of documents, you specify which elements and attributes are allowed in a complying XML document, and how those elements and attributes are related to each other. In the XML-Data Reduced schema, specifying an *ElementType* element and an *AttributeType* element defines the elements and attributes, respectively. You can then declare an instance of an element or an attribute using element or attribute element tags.

The schema must begin with the *Schema* element that must include the following namespace.

```
urn:schemas-microsoft-com:xml-data
```

To use XDR schema data types, the Schema element must be specified like this:

```
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
 xmlns:dt="urn:schemas-microsoft-com:datatypes">
```

Next, the content model of the elements and attributes that make up an XML document must be specified. The content model describes the content structure of elements and attributes in the XML document. The *model*, *minOccurs*, *maxOccurs*, *order*, *content*, *minLength*, *maxLength*, *default*, and *type* attributes can be used to qualify the declaration elements that are used to define elements and attributes and describe their content structures. These declaration elements are:

- *ElementType*: Assigns a type and condition to an element and what, if any, child elements it can contain.

- *AttributeType*: Assigns a type and condition to an attribute.
- *Attribute*: Declares that an instance of a previously defined *AttributeType* can appear within the scope of the named *ElementType* element.
- *Element*: Declares that an instance of a previously defined *ElementType* can appear within the scope of the named *ElementType.*

The content of the schema begins with the definitions of the innermost *AttributeType* and *ElementType* elements. Our example schema begins by declaring the elements and attributes of the *ORDERLINE* element that is the innermost element in the document.

```
<AttributeType name="ID" dt:type="string" required="yes" />
<AttributeType name="LINE" dt:type="int" />
<ElementType name="PRICE" dt:type="number" />
<ElementType name="QUANTITY" dt:type="number" />
<ElementType name="PRODUCT" content="textOnly" />
```

The set of allowable attributes for describing the content model of an *ElementType* is listed in **Table 3**. Keep in mind that the attribute names and values are case-sensitive and must be specified exactly as listed in the table.

***Table 3**. Allowable attributes for the ElementType element.*

Name	Description	Allowable values
content	An indicator of whether the content must be empty or can contain text, elements, or both. The default value is mixed.	empty: cannot contain content textOnly: can contain only text and not other elements eltOnly: can contain only elements and no text mixed: can contain a mixture of named elements and text
dt:type	The data type of the element. We have listed some of them. For a complete listing, refer to the XDR Schema Data Types in the XDR Schema Reference section of the MSXML4 SDK.	boolean: 1=true and 0=false date: date in ccyy-mm-dd format datetime: date and time in ccyy-mm-ddThh:mm:ss format int: integer number: a number with no limit on the digits string: a character string
model	Indicates whether the content must include only what is defined in the schema. The default is open.	open: can include additional elements or attributes not declared explicitly in the content model closed: can include only what is specified in the content model
Name	The name of the element. This attribute is required.	
Order	The order in which the elements appear.	one: only one of the specified set of elements is permitted. When order=”one”, the content model must be specified as closed. seq: the elements must appear in the specified sequence. many: the element may or may not appear in any order. When order=”many”, minOccurs and maxOccurs attributes no longer apply during validation.

After an *ElementType* has been defined, it can be used to declare an instance of that *element* within another *ElementType* definition. For example, price, quantity, and product have already been defined. We can now use these items as *elements* to define the *ORDERLINE ElementType* like this:

```
<ElementType name="ORDERLINE" content="mixed">
  <attribute type="LINE" />
  <attribute type="ID" />
  <element type="PRICE" />
  <element type="QUANTITY" />
  <element type="PRODUCT" />
</ElementType>
```

The *Element* element has the following attributes:

- *type*: The name of an *ElementType* element that has already been defined.
- *minOccurs*: The minimum number of times this element can appear. When this value is 0, the element is optional. The default value is 1.
- *maxOccurs*: The maximum number of times this element can appear. When this value is "*" the element can occur an unlimited number of times. The default value is 1. So by default, declared elements must occur once and only once within their parents.

This information enables you to begin authoring simple XDR schemas. For more information, refer to the MSXML4 SDK.

How do I create an XSD schema? (*Example: OrderSchema.xsd*)

The XML Schema definition language (XSD) enables you to define the structure and data types for XML documents. These definitions conform to the World Wide Web Consortium (W3C) 2001 recommendation specifications. MSXML4 offers support for XSD schemas. If you are using an earlier version of the parser, you do not have a choice between XSD and XDR because the earlier versions only support XDR schemas.

In addition to its built-in data types (such as integer, string, and so on), XML Schema also allows for the definition of new data types using the *simpleType* and *complexType* elements. A *simpleType* defines a value that can be used as content for an attribute or an element. This data type cannot contain elements or have attributes. *ComplexTypes* define elements that contain attributes and other elements. These two elements are the basic building blocks used for authoring XSD schema but there are many more. Consult the "XML Schema Elements" section of the XSD Schema Reference for a complete list.

An XSD schema must begin with the *schema* element that must include the following namespace:

```
http://www.w3.org/2001/XMLSchema
```

The content model of elements in an XSD schema can be specified using the same *type*, *minOccurs*, and *maxOccurs* attributes that are used in XDR schemas. However, the allowable

values in an XSD schema are different! For example, to specify an unlimited number of occurrences, the *maxOccurs* attribute must be specified as "unbounded" and not as "*". The data types are also different. In an XSD schema, integers are specified as "integer" instead of "int". Decimal values in XSD schemas are not specified as "number" as they are in XDR schemas. In an XSD schema the value "decimal" is used to denote this data type.

How do I define a simple type? (*Example: Fruits.xsd and Fruits.xml*)
Simple types are defined by deriving them from built-in data types and existing simple types. A simple type cannot contain elements and cannot have attributes. If a *simpleType* declaration has the schema element as its parent, it has global scope within that schema. Otherwise, its scope is limited to the *complexType* in which it is declared as an element. Simple types can be defined in one of the following ways:

- *restriction*: Restricts the range of values to a specific subset of values.
- *list*: Defines a space delimited list of values.
- *union*: Defines a *simpleType* that contains a union of the values of two or more inherited *simpleTypes*.

We can use *restriction* to specify the constraints on the set of allowable values for a *simpleType* in great detail. For example, we can define a *simpleType* called "Freezing2Boiling" to ensure that the values are in the range between 32 and 212. The *restriction* element's "base" attribute is required. Its value must be either a built-in data type or a *simpleType* element that is defined in this schema.

```
<xs:simpleType name="Freezing2Boiling">
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:minInclusive value="32"/>
    <xs:maxInclusive value="212"/>
  </xs:restriction>
</xs:simpleType>
```

We can also restrict the allowable values to a very specific set of values by creating a *simpleType* definition that is an *enumerated* type. For example, we can define a *simpleType* called "ShirtSize" that limits the allowable values to "small," "medium," and "large" like this:

```
<xs:simpleType name="ShirtSize">
  <xs:restriction base="xs:string">
    <xs:enumeration value="small"/>
    <xs:enumeration value="medium"/>
    <xs:enumeration value="large"/>
  </xs:restriction>
</xs:simpleType>
```

Another way to exercise control over the values permitted in the XML document is to define a *simpleType* as a *list* of values separated by white space. Defining lists of specific values, as in the following example, is a three-step process. First, the allowable values for the list must be enumerated as a *simpleType*. For example, we can define a *simpleType* called "Fruits" like this:

```
<xsd:simpleType name='Fruits'>
  <xsd:restriction base='xsd:string'>
    <xsd:enumeration value='apples'/>
    <xsd:enumeration value='oranges'/>
    <xsd:enumeration value='pears'/>
    <xsd:enumeration value='bananas'/>
    <xsd:enumeration value='grapes'/>
    <xsd:enumeration value='cherries'/>
  </xsd:restriction>
</xsd:simpleType>
```

Next, we must define a *simpleType* that is a *list* with an *itemType* of Fruits. This defines a data type that consists of a space separated list of values from the enumeration specified in the Fruits *simpleType* defined previously.

```
<xsd:simpleType name='ListOfFruits'>
  <xsd:list itemType='Fruits'/>
</xsd:simpleType>
```

Finally, if we want additional constraints, such as the maximum number of elements allowed in the list, we must define a third *simpleType* that derives from ListOfFruits and specify them here. The reason for this is simple: The parent node of a *restriction* element must be a *simpleType*, it cannot be a *list* node.

```
<xsd:simpleType name='FruitElement'>
  <xsd:restriction base='ListOfFruits'>
    <xsd:maxLength value='3'/>
  </xsd:restriction>
</xsd:simpleType>
```

This definition specifies that an element of type FruitElement can appear in the XML document and that it must consist of a maximum of three items from the enumeration list. These items must be separated by spaces.

Finally, we can define a *simpleType* as a collection of values from the specified simple data types. We begin by defining the simple data types of the member types. For example, we can define a simple data type for font size numbers like this:

```
<xsd:simpleType name="FontSizeNumber">
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:maxInclusive value="72"/>
  </xsd:restriction>
</xsd:simpleType>
```

We can define another simple data type for font size strings like this:

```
<xsd:simpleType name="FontSizeString">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="small"/>
    <xsd:enumeration value="medium"/>
    <xsd:enumeration value="large"/>
  </xsd:restriction>
</xsd:simpleType>
```

We can then define an attribute named "FontSize" as a simple data type that is a union of these two simple data types. This allows a conforming XML document to specify its FontSize attribute either as a number or as a string.

```
<xsd:attribute name="FontSize">
  <xsd:simpleType>
    <xsd:union memberTypes="FontSizeNumber FontSizeString" />
  </xsd:simpleType>
</xsd:attribute>
```

How do I define a complex type? (*Example: OrderSchema.xsd and OrderSchema2.xsd*)
A *complexType* is a type definition for an element that may contain attributes and elements. It defines the structure, content, and attributes of that element. A *complexType* can contain one and only one of the following elements, which determines the type of content allowed in the *complexType*.

- *simpleContent*: The complex type has character data or a *simpleType* as content and contains no elements, but may contain attributes. For example, to define an element that looks like this in our XML document:

```
<FruitList FontSize="small">cherries oranges apples</FruitList>
```

  We could use the FruitElement and FontSize *simpleTypes* that we defined in the preceding section to build a *complexType* called "SizedFruits" like this:

```
<xsd:complexType name="SizedFruits">
  <xsd:simpleContent>
    <xsd:extension base="FruitElement">
      <xsd:attribute name="FontSize">
        <xsd:simpleType>
          <xsd:union memberTypes="FontSizeNumber FontSizeString" />
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

  The *extension* element in this definition extends the FruitElement *simpleType* by adding the FontSize *attribute.* This process is analogous to creating a subclass in Visual FoxPro and giving the subclass some new properties.

- *complexContent*: Contains extensions or restrictions on a complex type that contains mixed content or elements only. For example, in ORDERSCHEMA2.XSD, we defined the *complexType* OrderLineType like this:

```
<xsd:complexType name="OrderLineType">
  <xsd:sequence>
    <xsd:element name="PRICE" minOccurs="1" maxOccurs="1" type="xsd:decimal" />
```

```
    <xsd:element name="QUANTITY" type="xsd:decimal" />
    <xsd:element name="PRODUCT" />
  </xsd:sequence>
  <xsd:attribute name="LINE" use="required" type="xsd:integer" />
  <xsd:attribute name="ID" use="required" />
</xsd:complexType>
```

We could create another complex data type called "ClearanceItemType" that extends the OrderLineType by adding some new elements like this:

```
<xsd:complexType name="ClearanceItemType">
  <xsd:complexContent>
    <xsd:extension base="OrderLineType">
      </xsd:sequence>
        <xsd:element name="DISCOUNT" type="xsd:decimal" />
        <xsd:element name="RETURNABLE" type="xsd:boolean" />
     </xsd:sequence>
    </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

- *group*: The *complexType* contains the elements defined in the referenced group. For example, we can redefine the OrderLineType data type using the *group* element like this:

```
<xsd:element name="PRICE" type="xsd:decimal" />
<xsd:element name="QUANTITY" type="xsd:decimal" />
<xsd:element name="PRODUCT" type="xsd:string" />
<xsd:attribute name="LINE" type="xsd:integer" />
<xsd:attribute name="ID" />

<xsd:group name="LineGroup">
  <xsd:sequence>
    <xsd:element ref="PRICE" />
    <xsd:element ref="QUANTITY" />
    <xsd:element ref="PRODUCT" />
  </xsd:sequence>
</xsd:group>

<xsd:complexType name="OrderLineType">
  <xsd:group ref="LineGroup" />
  <xsd:attribute ref="LINE" use="required" />
  <xsd:attribute ref="ID" use="required" />
</xsd:complexType>
```

- *sequence*: All the elements must be appear in the specified order in the document.
- *choice*: One and only one of the elements must appear inside the containing element.
- *all*: The elements in the group are all optional and may appear in any order inside the containing element.

As you can see, using XSD schemas is extremely powerful and gives you lots of control over validating your XML documents. The information presented here barely scratches the surface. It would be impossible to provide a complete and exhaustive schema reference in the space of a few pages, but the information we have provided is enough to get you started using schemas and to make sense of the documentation provided in the SDK.

How do I use the SchemaCache to validate XML documents?

The DOM's *SchemaCache* provides a simple mechanism by which XML documents can be validated quickly. Once a schema is loaded into the cache, it can be used to validate multiple documents when they are loaded. MSXML4 also exposes a Schema Object Model (SOM) that can be used when finer control is required over the validation process. However, a discussion of the SOM is beyond the scope of this section, but the MSXML SDK provides excellent documentation on its use along with a tutorial that illustrates how to walk the SOM.

The first step in validating an XML document against a schema is to instantiate the DOM like this:

```
loParser = CREATEOBJECT( 'MSXML2.DomDocument.4.0' )
```

Next, the *SchemaCache* must be instantiated:

```
loSchemaCache = CREATEOBJECT( 'MSXML2.XMLSchemaCache.4.0' )
```

and the schema must be added to the cache:

```
loSchemaCache.Add( '', 'OrderSchema.xml' )
```

The first argument is the namespace to associate with the specified schema. When the empty string is passed, the schema is associated with the empty namespace, xmlns="".

After the schema is added to the cache, the parser's *schemas* property must be set to point to the *SchemaCache* so the parser knows what schema is to be used to validate the XML document. When the parser's *ValidateOnParse* property is set to its default value, which is true, the document is validated during parsing; that is, when it is loaded. After the document is loaded, the DOM's *ParseError* object can be examined like this to determine whether or not the document is valid:

```
IF loParser.ParseError.ErrorCode # 0
  MESSAGEBOX(loParser.ParseError.Reason,16,'Unable to validate XML Document')
ENDIF
```

If we remove the ID attribute from the **<CUSTOMER>** node of the ORDERS.XML file we generated using the sample form, GENERATEXML.SCX, and run the program SCHEMATEST.PRG, the message box shown in **Figure 3** is displayed to notify us that the XML is not valid.

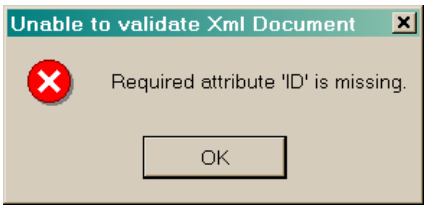


***Figure 3**. Reason that XML document is not valid.*

We can also use the node's *Definition* property to access the definition for that node in the associated schema. For example, to view the definition of the **<CUSTOMER>** node in ORDER.XML, we use this code:

```
loCust = loParser.DocumentElement.FirstChild
loDef = loCust.Definition
?loDef.xml
```

And this is what is displayed on the screen:

```
<ElementType xmlns="urn:schemas-microsoft-com:xml-data" name="CUSTOMER"
 content="mixed">
  <attribute type="ID"/>
  <element type="COMPANY" minOccurs="1" maxOccurs="1"/>
  <element type="CONTACT" minOccurs="1" maxOccurs="1"/>
  <element type="ORDERHEADER" minOccurs="1" maxOccurs="*"/>
</ElementType>
```

What is XSLT?

eXtensible Stylesheet Language Transformations (XSLT) evolved from the early eXtensible Stylesheet Language (XSL) standard. XSL is an XML-based language designed to transform an XML document into another XML document or into some other text format such as HTML. XSLT is a declarative language. This means that when we use XSLT, we specify how we want the final result to look but we do not specify *how* the source document should be transformed to obtain this result. This is the job of the XSL processor.

As a programming language, XSLT has support for:

- A set of flexible data types: Boolean, number, string, node-set, and external objects
- A set of operations such as **<xsl:template>**, **<xsl:apply-templates>**, **<xsl:sort>**, **<xsl:output>**
- Programming flow-control such as **<xsl:if>**, **<xsl:for-each>**, **<xsl:choose>**

XSLT enables you to define templates that contain the rules for formatting the output from the XML source document. A template is the content of an **<xsl:template>** element in the stylesheet and consists of two components: a match pattern and the template itself. The match pattern consists of a match attribute and an expression, which specifies which portions of the source tree are processed by the template rule. For example, the following match pattern processes all **<CUSTOMER>** elements in our sample document, ORDERS.XML:

```
<xsl:template match="CUSTOMER">
```

Each time the template locates something in the source tree that matches the pattern, it places content in the result tree. In other words, the template rule instantiates the template for each match. The content of a template depends on how the template transforms the source document. For example, if the output is formatted HTML, the template might consist of HTML markup and scripts.

The XSL pattern language provides the syntax for traversing the tree structure of an XML file, so a good starting point for a discussion of XSLT is a brief look at XSL patterns.

What are XSL patterns?

The purpose of a pattern is to restrict the set of candidate nodes in a node-set to just those nodes that meet a particular condition, or set of conditions. The most common use of a pattern is in the *match* attribute of an **<xsl:template>** element of an XSLT document where the pattern specifies the nodes to which the template rule is applied.

Patterns are defined in terms of the name, type, and string-value of a node and to that node's relative position to other nodes in the tree. Several examples of XSL patterns are listed in **Table 4**.

***Table 4**. XSL pattern examples.*

Pattern	Matches
/	The root node
node()	Any node other than an attribute node and the root node
text()	Any text node
*	Any element
@*	Any attribute
@class	Any class attribute (not any element that has a class attribute)
ORDERHEADER	Any ORDERHEADER element
PRODUCT|QUANTITY	Any PRODUCT element and any QUANTITY element
CUSTOMER/ORDERHEADER	Any ORDERHEADER element with a CUSTOMER parent
CUSTOMER//PRODUCT	Any PRODUCT element with a CUSTOMER element as an ancestor
//*[@ID='ALFKI']	The element with an ID attribute of 'ALFKI'
PRODUCT[1]	Any PRODUCT element that is the first PRODUCT child element of its parent
PRODUCT[position() mod 2 = 1]	Any PRODUCT element that is an odd-numbered PRODUCT child of its parent

It is difficult to talk about XSL patterns without at least mentioning XPath because the two are so closely related. The reason is that XPath was developed in order to provide a common syntax for the functionality shared by XSLT and XPointer (another language that specifies constructs for addressing the internal structures of XML documents). All XSL patterns are XPath expressions, although the reverse is not true. An in-depth discussion of XPath expressions is beyond the scope of a single section in a book. However, the MSXML SDK contains a detailed *XPath Reference* and *XPath Developer's Guide* that provide excellent detailed information on the subject.

What XSLT elements do I use to define my template?

The XSLT transformation processor merges data from the XML source document with the template. XSLT templates are defined using a small set of XML elements (see **Table 5**). These elements may represent either XSL processing instructions or data. An XSL instruction is any element that is used in a template body. An element defined as a top-level element must be a child of the `<xsl:stylesheet>` element.

***Table 5**. XSLT elements.*

Element name	Usage
xsl:apply-imports	An XSL instruction that is used in conjunction with imported stylesheets to augment the template rule in the current stylesheet.
xsl:apply-templates	An XSL instruction that selects a set of nodes for processing by applying the matching template rule. The body of each template declares which nodes it is interested in instead of having the template rule for the parent node describe in detail how each of its children should be processed.
xsl:attribute	Creates an attribute node and attaches it to an output element.
xsl:attribute-set	A top-level element that defines a named set of attributes, providing a way to define commonly used attributes in a single place that can then be applied as a whole to any output element.
xsl:call-template	An XSL instruction, analogous to a procedure call that invokes a template by name.
xsl:choose	An XSL instruction that functions like a CASE statement in Visual FoxPro. It provides multiple conditional testing in conjunction with the <xsl:otherwise> element and <xsl:when> element.
xsl:comment	An XSL instruction that writes a comment to the current output location.
xsl:copy	An XSL instruction that copies the current node from the source to the output. It copies only the current node and does not copy any of its children.
xsl:copy-of	An XSL instruction that inserts subtrees and result tree fragments into the result tree. In other words, it copies a node and all of its descendants to the current output location.
xsl:decimal-format	A top-level element that declares a decimal-format, which controls the interpretation of a format pattern used by the format-number function.
xsl:element	An XSL instruction that creates an element with the specified name in the output.
xsl:fallback	An XSL instruction that calls template content that can provide a reasonable substitute to the behavior of the new element when encountered.
xsl:for-each	An XSL instruction that selects a set of nodes using an XPath expression and applies a template repeatedly to each node in a set.
xsl:if	An XSL instruction that works just like the IF statement in Visual FoxPro. It allows simple conditional template fragments.
xsl:import	Imports another XSLT file.
xsl:include	This is analogous to using an include file in Visual FoxPro. Definitions of standard elements that are used in many stylesheets can be defined in a single XSLT file and incorporated into a stylesheet without change. If it is necessary to override any of these definitions, use <xsl:import> instead.
xsl:key	A top-level element that declares a named key for use with the key () function in XML Path Language (XPath) expressions.
xsl:message	An XSL instruction that sends a text message to either the message buffer or a message dialog box and optionally terminates execution of the stylesheet.
xsl:namespace-alias	A top-level element that replaces the prefix associated with a given namespace with another prefix.
xsl:number	An XSL instruction that inserts a formatted number into the result tree.

***Table 5**. Continued.*

Element name	Usage
xsl:otherwise	Provides multiple conditional testing in conjunction with the <xsl:choose> element and <xsl:when> element.
xsl:output	A top-level element used to control the format of the stylesheet output.
xsl:param	Declares a named parameter for use within an <xsl:stylesheet> element or an <xsl:template>element and allows specification of a default value. When used as a top-level element, the scope of the parameter is global and when used within a template the scope is local.
xsl:preserve-space	A top-level element that preserves white space in a document.
xsl:processing-instruction	An XSL instruction that generates a processing instruction in the output.
xsl:sort	Specifies sort criteria for node lists selected by <xsl:for-each> or <xsl:apply-templates>.
xsl:strip-space	A top-level element that strips white space from a document.
xsl:stylesheet	Specifies the document element of an XSLT file, containing all other XSLT elements.
xsl:template	A top-level element that defines a reusable template for generating the desired output for nodes of a particular type and context.
xsl:text	An XSL instruction that generates text in the output.
xsl:transform	Synonym for <xsl:stylesheet>.
xsl:value-of	An XSL instruction that inserts the value of the selected node as text into the result tree.
xsl:variable	Can be a both top-level element and an XSL instruction. It is used to declare global (when a top-level element) and local (when an XSL instruction) variables in a stylesheet.
xsl:when	Provides multiple conditional testing in conjunction with the <xsl:choose> element and <xsl:otherwise> element. It defines the condition to be tested and the action to be performed when the condition is true.
xsl:with-param	Passes a parameter to a template when calling a template using either <xsl:call-template> or <xsl:apply-templates>.

How do I use XSLT to transform my XML documents? (*Example: Orders.xsl and XSLTTest.prg*)

An XSLT processor is required to apply an XSLT stylesheet to an XML document and produce the transformed output. MSXML is only one of the XSLT processors that are available to us, but it is very convenient because it allows us to run XSLT stylesheets within Internet Explorer. However, there is one caveat here: The best way to ensure that the correct version of the XSLT processor is available is to install Internet Explorer version 6.

MSXML versions 2.6 and earlier only support the XSL standard, and this is the default processor for Internet Explorer 5.0 and 5.5. MSXML versions 3.0 and later support XSLT 1.0. So if you are still using IE5 or IE5.5, MSXML 3.0 must be installed in Replace mode so that it becomes the default XML/XSLT processor. However, as we mentioned earlier, running in Replace mode is not recommended because it may leave the system in an unstable state, so this is not a very good solution.

How do I construct my stylesheet?

We have included two sample stylesheets that transform ORDERS.XML, the file produced by our GenerateXML form earlier in this chapter, into formatted HTML. Why two samples? Because there are essentially two different methodologies that we can use to construct our stylesheets.

The first, exemplified in ORDER.XSL, is known as "pull" processing because the template pulls the nodes in and processes them itself. The second, used by ORDERS2.XSL, is known as "push" processing because when it processes the source document, it pushes the nodes out to be processed using **<apply-templates>**. This approach works very well when the output has essentially the same structure and sequence as the source document and all that is required is to format it for display.

Regardless of which methodology is used to construct the stylesheet, this must be the root node:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

This element can have a set of **<xsl:template>** elements representing different output templates. The next line:

```
<xsl:template match="CUSTOMER">
```

defines a template for the **CUSTOMER** element. The *match* attribute of this element contains a pattern expression that returns a node set containing the **<CUSTOMER>** node in the source document. The template is then applied to that node. We'll examine ORDERS.XSL to see how the stylesheet works.

The first part of the template constructs the preamble for the HTML page that we want to output.

```
<html>
  <head>
    <title>
      Using XSLT to transform the Orders XML document into formatted HTML
    </title>
  </head>
  <body>
```

Next, the value of the **<COMPANY>** node that is a child of the **<CUSTOMER>** node is inserted into the output.

```
    <h1>Orders for <xsl:value-of select="COMPANY"/></h1>
```

The *select* attribute of the **xsl:value-of** element is evaluated relative to the template's current node; in this case, the **<CUSTOMER>** node. This works well because, in our source document, each customer has a single **<COMPANY>** node as a child node and the **<COMPANY>** node does not have any children. If this expression had returned more than a single node, only the text of the first node would have been returned. Furthermore, if the **<COMPANY>** node had any children, the concatenated text nodes of the **<COMPANY>** element's sub-tree (with the markup removed) would have been returned.

Now we are ready to list each of the orders for this customer. This line:

```
    <xsl:for-each select="ORDERHEADER">
```

returns a node set that contains all of the **<ORDERHEADER>** nodes for the current customer. It is important to understand that when the processor begins to process a template, the source node associated with that template becomes the current node. This current node defines a context node for evaluating the remaining XPath expressions within the template. The point here is that XSLT template rules cannot be relied on to "fire" in any particular order; each one is evaluated strictly in terms of the context established at that point

You may have noticed that the **for-each** and **value-of** elements in our stylesheet both have a *select* attribute. The context node for both of these elements is the current node. In the case of the **xsl:value-of**, the context node does not change. In the case of the **xsl:for-each**, the *select* returns a node set, and each node in this set becomes the current node for further processing. So the following template is applied to each **<ORDERHEADER>** element to display the order number, date, and total amount of the order and begin an unordered list of the order lines that it contains.

```
      <h3>
        Order Number: <xsl:value-of select="@ID"/>
        Date: <xsl:value-of select="DATE"/>
        Total: <xsl:value-of select="AMOUNT"/>
      </h3>
      <ul>
```

Finally, another **xsl:for-each** instruction is used to iterate through the order lines for each of the **<ORDERHEADER>** elements to output a line of the list before adding all the required closing tags.

```
      <xsl:for-each select="ORDERLINE">
        <li>
          <xsl:value-of select="concat(QUANTITY, ' ')"/>
          <xsl:value-of select="PRODUCT"/>
        </li>
      </xsl:for-each>
      </ul>
    </xsl:for-each>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

The HTML produced by applying this stylesheet to ORDERS.XML appears in **Figure 4**.

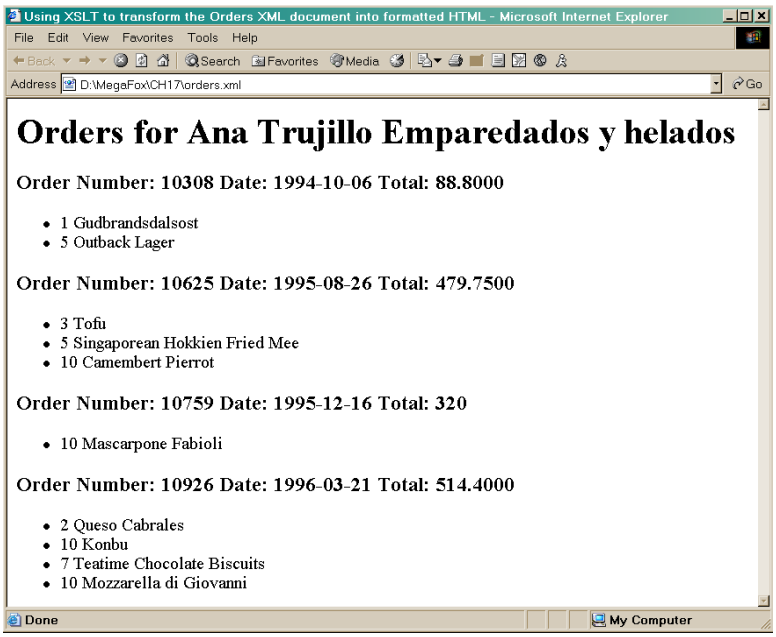


***Figure 4**. HTML formatted output obtained by applying Order.xsl to Orders.xml.*

Now let's take a brief look at ORDERS2.XSL to see how the same stylesheet could have been written using "push" processing. The key difference between this stylesheet and the previous one is that ORDERS2.XSL does not use the **`xsl:for-each`** instruction inside of a single template to iterate through the child nodes. Instead, a separate template is defined for each node-set that we want to process. The specified template is applied recursively using the **`xsl:apply-templates`** instruction.

There is no template for the root node defined in the stylesheet, so the built-in template is invoked to process all of the root node's children. The root node has only one child node, the **`<CUSTOMER>`** node, so its template is instantiated and some HTML is added to the result tree. The **`xsl:apply-templates`** instruction is then used to process the children of the **`<CUSTOMER>`** node.

```
<xsl:template match="CUSTOMER">
  <html>
    <head>
      <title>
        Using XSLT to transform the Orders XML document into formatted HTML
      </title>
    </head>
    <body>
      <xsl:apply-templates select="COMPANY"/>
      <xsl:apply-templates select="ORDERHEADER"/>
    </body>
  </html>
</xsl:template>
```

The template for the **<COMPANY>** node is very simple indeed. It merely added the company name to the result tree as formatted HTML.

```
<xsl:template match="COMPANY">
  <h1>Orders for <xsl:value-of select="."/></h1>
</xsl:template>
```

The template for the **<ORDERHEADER>** node adds the order header information to the output tree. It then uses the **xsl:apply-templates** instruction to process all of its children, the order lines.

```
<xsl:template match="ORDERHEADER">
  <h3>
    Order Number: <xsl:value-of select="@ID"/>
    Date: <xsl:value-of select="DATE"/>
    Total: <xsl:value-of select="AMOUNT"/>
  </h3>
  <ul>
    <xsl:apply-templates select="ORDERLINE"/>
  </ul>
</xsl:template>
```

Finally, the template for the order lines adds each order line to the result tree.

```
<xsl:template match="ORDERLINE">
  <li>
    <xsl:value-of select="concat(QUANTITY, ' ')"/>
    <xsl:value-of select="PRODUCT"/>
  </li>
</xsl:template>
```

We can now use a stylesheet processing instruction in the XML source document to link it to our XSLT file. This instruction must be at the beginning of the XML file following the XML declaration:

```
<?xml-stylesheet type="text/xsl" href="orders.xsl" ?>
```

Now, when the XML document is opened in Internet Explorer, we see the formatted HTML (Figure 4) instead of the XML (Figure 1).

How do I use the DOM's XSL processor to transform XML?

MSXML's XSL processor can be used to transform a source document into a different format without explicitly linking the source document to a given stylesheet. This is useful when many different transformations are required for a single XML document. It can also be used to generate dynamic HTML content from XML on the fly to create a Web page.

The first thing that we need to do is to load both the XML source document and the stylesheet that contains the rules for the transformation. In order to increase performance, we also create an *XSLTemplate* object to cache the compiled XSLT stylesheet and use this object to perform the transformation.

```
loXslt = CREATEOBJECT( 'MSXML2.XSLTemplate.4.0' )
loParser = CREATEOBJECT( 'MSXML2.DOMDocument.4.0' )
loXslDoc = CREATEOBJECT( 'MSXML2.FreeThreadedDOMDocument.4.0' )
```

This code loads the stylesheet:

```
loXslDoc.Async = .F.
loXslDoc.Load( "Orders.xsl" )
loXslt.Stylesheet = loXslDoc
```

And this code loads the source document:

```
oxmlDoc.async = .f.
loParser.Load( "Orders.xml" )
```

The only thing left to do is to create an instance of the XSLT processor and apply the transformation:

```
loXslProc = loXslt.CreateProcessor()
loXslProc.Input = loParser
loXslProc.Transform()
```

At this point, the transformed result is stored in the *Output* property of the XSLT processor. If we want to, we can save the transformed result as a file like this:

```
STRTOFILE( loXslProc.Output, 'Orders.html' )
```

Conclusion

XML is sometimes called the ASCII of the Web, and it is definitely here to stay. If you are not yet working with it, don't worry. You will be in the near future, especially since Visual FoxPro 7.0 creates and consumes Web Services. There is so much information available about XML and XSLT that entire books have been devoted to each of these topics. Obviously, we cannot hope to cover them in depth in the space of a single chapter. However, we have tried to provide a good overview that will enable you to make sense of some of the resources that are available on these topics.

What is ADO?

ADO is another TLA (Three-Letter Acronym) that stands for ActiveX Data Objects. It provides a consistent set of interfaces for accessing data by using the services of some OLE DB provider. It is OLE DB that actually provides access to data, but ADO makes it much easier for us to work with OLE DB.

Although most applications now understand XML, many older applications do not. For example, if you need to send data to an Office 97 application, you cannot send XML. If you want to push the data out to that application, the data needs to be dished up as an ADO RecordSet. Working with ADO is not difficult at all and many of the concepts may seem familiar because ADO is based on the Visual FoxPro cursor engine.

As is the case with the XML portion of this chapter, this section is not meant to be a comprehensive discussion of ADO and all of its details. Instead, we hope to give you a broad overview of its object model and show you how to tap into some of its functionality to give you a good starting point for future explorations of the subject. Entire books have been written about ADO, so a complete discussion of it is clearly beyond the scope of a single section in just one chapter of a book.

The ADO object model

The first step to using ADO in an application is to understand its object model (see **Figure 5**). In this section we will discuss the different ADO objects and what they do.

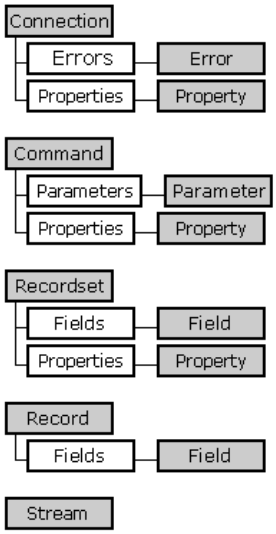


***Figure 5**. ADO objects and their collections.*

How do I use the Connection object?

The ADO Connection object manages the communication between the application and the database. After the Connection object is instantiated, you can access its data store by setting a couple of properties and invoking its *Open()* method. This code creates the Connection object:

```
oConnection = CREATEOBJECT( 'ADODB.Connection' )
```

Now we are ready to tell the connection which OLE DB provider to use and which database to access. We can open the database that ships with the Visual FoxPro sample application by setting its *ConnectionString* property and then invoking its open method like this:

```
lcstr = 'provider=vfpoledb.1; data source=' + HOME(2) + 'data\testdata.dbc'
oConnection.ConnectionString = lcstr
oConnection.Open()
```

Alternatively, we can accomplish the same thing like this:

```
oConnection.Provider = 'vfpoledb.1'
lcstr = HOME(2) + 'data\testdata.dbc'
oConnection.Open( lcstr )
```

The Connection object has several methods that allow you to manage changes to the database inside transactions. *BeginTrans*, *CommitTrans*, and *RollbackTrans* all perform the same function as their similarly named counterparts in Visual FoxPro. In addition to the transactional methods, the following are likely to be of most interest to you:

- *Execute()*: Used to submit queries including queries that generate RecordSets. It can be used to issue an action query to modify data, manipulate objects, or change database settings.

  So, to obtain a RecordSet from the testdata database that contains all of the customers in the customer table, we can use the connection's *Execute()* method like this:

  ```
  lcSql = "select * from customer"
  oRs = oConnection.Execute( lcsql )
  ```

- *OpenSchema()*: Creates an ADO RecordSet containing information about tables, fields, and so on. It is analogous to Visual FoxPro's `DBGETPROP()` function.
- *Close()*: Closes an open Connection object.

The Connection object also has an *Errors* collection that consists of one or more error objects. We can write code in our application's global error handler or our form's *Error()* method to trap and handle any errors that are reported to us by ADO. All we need to do is trap for error 1429 ("OLE error") and interrogate the connection's *Errors* collection to find out what the problem is.

How do I use the Command object?

The purpose of the ADO Command object is, as its name implies, to run commands. It allows you to execute queries that retrieve data as well as those that update the database. You can also use the Command object to call stored procedures and inspect any return values. Return values are handled by the Command object's *Parameters* collection.

The Command object has the following properties:

- *ActiveConnection*: The data store against which to execute commands. This can be an object reference to an existing Connection object or it can be a connection string. If this property is set to a connection string, ADO creates a new Connection object behind the scenes and attempts to connect to the database using it.

- *CommandText*: Usually contains the query string to be executed, but can also be used to call stored procedures.
- *CommandTimeOut*: Specifies the number of seconds that ADO waits for the query to complete before it times out and cancels the query. The default is 30 seconds, but if you want the query to run indefinitely without timing out, just set this property to 0.
- *CommandType*: Used to optimize the execution of the *CommandText* property. Allowable values and their meanings are listed in **Table 6**.

***Table 6**. Allowable value for the command object's CommandType property.*

Constant	Value	Description
adCmdText	1	The query will not be modified by ADO.
adCmdTable	2	ADO will insert "SELECT * FROM" in front of the query specified in the *CommandText* property.
adCmdStoredProc	4	ADO will format the query specified in the *CommandText* property as a call to a stored procedure.
adCmdUnknown	8	This is the default value. It means that ADO will try different methods of executing the query until the query executes successfully.
adCmdFile	256	Indicates that the *CommandText* property refers to a file name. This value is not allowed as a *CommandType* for a Command object. It can only be used by the *Open()* and *Requery()* methods of a RecordSet object.
adCmdTableDirect	512	Tells ADO to use an advanced set of OLE DB API calls to retrieve all the records in the table name specified in *CommandText*. This value is not allowed as a *CommandType* for a Command object. It can only be used by the *Open()* method of the RecordSet object.

- *Name*: Name of the Command object.
- *Prepared*: Indicates whether the OLE DB provider should save a compiled version of a command before execution to optimize subsequent execution.
- *State*: Indicates whether the command is still executing (if it was executed asynchronously) or if execution has finished.

The Command object also supports the following methods:

- *Cancel()*: Cancels an asynchronous query.
- *CreateParameter()*: Creates a parameter object for the Command object's parameters collection. This method supports the following parameters:
  - *Name:* Optional parameter specifying the name of the parameter object.

  - *Type:* The parameter's data type. Must be a valid ADO DataTypeEnum value. These are listed in the include file, ADOVFP.H that is included with the sample code for this chapter.
  - *Direction:* Optional parameter that specifies whether this is an input or output parameter.
  - *Size:* Optional parameter that specifies the maximum size of the parameter.
  - *Value:* A variant that specifies the value of the parameter.
- *Execute()*: Executes the query specified by the *CommandText* property.

This example creates a Command object and sets it up to execute a parameterized query against the customers table in the sample application that ships with Visual FoxPro.

The first thing that we need to do is to tell the Command object where to find the data source. If we have a Connection object handy, all we need to do is set the Command object's *ActiveConnection* property like this:

```
oCmd = CREATEOBJECT( 'ADODB.Command' )
oCmd.ActiveConnection = oConnection
```

If we do not have an open connection yet, we can simply set the Command object's *Active Connection* to a connection string like this:

```
lcstr = "provider=vfpoledb.1; data source=" + HOME(2) + "data\testdata.dbc"
oCmd.ActiveConnection = lcStr
```

Next, we set the properties of the Command object so that it knows what to do:

```
WITH oCmd
  .CommandText = "SELECT * FROM Customer WHERE Country = ?"
  .CommandType = 1     &&adCmdText
ENDWITH
```

Before we can call the Command object's *Execute()* method, we must add a Parameter object to the Command object's *Parameters* collection that contains the name of the country for which to execute the query.

How do I use the Command object's Parameters collection?

The Command object exposes a *Parameters* collection that you can use to run parameterized queries. If you are using the Command object to call a stored procedure, the *Parameters* collection is used to pass arguments to, and accept return values from, the stored procedure.

You can add a Parameter object to the collection by invoking the Command object's *CreateParameter()* method with the correct arguments, or you can use `oParm = CREATEOBJECT( 'ADODB.Parameter' ).` After the parameter has been created and its required properties have been set, use the *Append()* method of the *Parameters* collection to add the parameter to the collection. This example creates a Parameter object that contains the name of a country and adds it to the Command object's *Parameters* collection:

```
oParm = oCmd.CreateParameter()
WITH oParm
  .Name = "oCountryParm"
  .Type = 12      && adVariant
  .Direction = 1  && input parameter only
  .Size = 7
  .Value = 'Germany'
ENDWITH
oCmd.Parameters.Append( oParm )
```

Now that we have set up the parameter, we are ready to run the query by invoking the Command object's *Execute()* method like this:

```
oRs = oCmd.Execute()
```

If we now require a list of all the customers in France, all we need to do is reset the size and the value of the parameter like this and re-execute the query:

```
oCmd.Parameters( 'oCountryParm' ).Size = 6
oCmd.Parameters( 'oCountryParm' ).Value = 'France'
oRs = oCmd.Execute()
```

The OLE DB provider for Visual FoxPro behaves differently than the providers for SQL Server and Access when it comes to executing stored procedures. In SQL Server and Access, when you can call a stored procedure, the results are returned as an ADO RecordSet. Calling a stored procedure using the OLE DB provider for Visual FoxPro does not do this. Actually, to be absolutely correct about it, executing a stored procedure does return a RecordSet, just not the kind of RecordSet you might be expecting. The RecordSet that is returned from a Visual FoxPro stored procedure contains a single field called *Return_Value*. You cannot return a cursor in the form of an ADO RecordSet by executing a stored procedure in Visual FoxPro. So if you need to return a single value, the ability to execute a stored procedure using ADO may be of some limited usefulness. However, we do not think that this is useful enough to deserve much more than the passing mention we have just given it.

How do I use the RecordSet object?

Just as you can think of the Connection object as your link to the database, you can think of the RecordSet object as your link to its data. When you submit a query to the database using ADO, the result is stored in a RecordSet object. You can then examine the RecordSet for the results of the query. This object also supports other functionality such as updating, sorting, and filtering.

The RecordSet object has a large number of properties, many of which, as Visual FoxPro developers, we are not going to be terribly interested in. Generally speaking, we are going to use ADO RecordSets to transfer information back and forth between application boundaries. If our Visual FoxPro application must process an ADO RecordSet that it has received from another application, we are going to convert it to a Visual FoxPro cursor before processing it. So, in our opinion, the RecordSet properties that are concerned with navigation through the RecordSet (for example, the *BookMark* property) are only of academic interest and they are not listed here. These are the properties that you need to know about when sending RecordSets back and forth between applications:

- *ActiveCommand*: An object reference to the Command object that created the RecordSet. This property is read-only and is available even after the RecordSet is closed.
- *ActiveConnection*: For an open RecordSet, this contains an object reference to the Connection object that was used to retrieve the data. This property does not change when the RecordSet is closed, but the only time you can modify it while the RecordSet is open is if the RecordSet is a client-side RecordSet.
- *BOF and EOF*: Does the same thing as the Visual FoxPro `BOF()` and `EOF()` functions.
- *CursorLocation*: Determines how the results of the query will be stored.
  - adUseServer (1) is the default. This *CursorLocation* means that the OLE DB provider or the database manages the query results.
  - adUseClient (2) means that the ADO cursor engine manages the query results.
- *CursorType*: Contains one of the values listed in **Table 7**.

***Table 7**. CursorTypeEnum values.*

Constant	Value	Description
adOpenForwardOnly	0	Default for server-side RecordSets. You can only scroll forward in this type of RecordSet and cannot scroll backward.
adOpenStatic	3	Default for client-side RecordSets. Supports both forward and backward scrolling. Changes made by other users are not visible.
adOpenKeyset	1	Supports scrolling in both directions and modifications and deletions (not inserts) made by other users are visible.
adOpenDynamic	2	Supports scrolling in both directions and all changes, including inserts, made by other users are visible.

- *Fields*: The *Fields* collection is the default property for RecordSet objects, and the *Value* property is the default property for the Field object. This means that in VB, you can refer to a field in the collection without explicitly including *Fields* in the hierarchy like this: `oRs( 0 ).Value` or even just `oRs(0)` since *Value* is the default property for the Field object. However, it does not work like this in Visual FoxPro. This zero-based collection can be accessed by using either the Field's index or its name. For example, you can access the value of the *Cust_ID* field in a RecordSet created from the Customers table in the Testdata database like this:

```
oRs.Fields( "Cust_ID" ).Value
oRs.Fields( 0 ).Value
```

- *MaxRecords*: Limits the number of records returned by the query.
- *RecordCount*: The number of records in the RecordSet. This property contains –1 if the provider or the cursor does not support *Bookmarks*.
- *Source*: Contains information about the query used to build the RecordSet. This can be either a SQL statement or an object reference to a Command object.
- *State*: Contains one of the values from **Table 8** to indicate the current state of the RecordSet.

***Table 8**. ObjectStateEnum values.*

Constant	Value	Description
adStateClosed	0	The RecordSet object is closed.
adStateOpen	1	The RecordSet object is open.
adStateConnecting	2	Not applicable to the RecordSet object.
adStateExecuting	4	The RecordSet object is executing the specified query.
adStateFetching	8	The RecordSet object is fetching the query results.

The RecordSet object has a number of methods that allow us to manipulate it. Many of these methods are used for navigating through the RecordSet and have Visual FoxPro counterparts such as `SEEK()` and `LOCATE`. We have omitted these methods because they are only of academic interest to us as Visual FoxPro developers. Obviously, if we must accept an ADO RecordSet from another application for processing, we are going to convert it to a Visual FoxPro cursor first. With this in mind, the most important methods of the RecordSet object are:

- *AddNew()*: Inserts new data into the RecordSet. Analogous to **APPEND BLANK**. This method accepts two parameters:
  - *FieldList:* An array containing the names of the fields to be populated.
  - *Values:* An array of values for the fields in the *FieldList* array.

  The field values can be populated individually like this:

```
oRs.AddNew()
oRs.Fields( "Cust_ID" ).Value = "DEWEY"
oRs.Fields( "Company" ).Value = "Dewey Cheetum and Howe"
oRs.Fields( "Contact" ).Value = "Seymour Cash"
```

  or we could create two arrays, one with the field names and one with the field values (in the correct order!) and pass these arrays as arguments to the *AddNew()* method like this:

```
oRs.AddNew( laFields, laValues )
```

- *Cancel()*: Terminates execution of an asynchronous query.
- *Clone()*: Creates a new RecordSet object from the current RecordSet.
- *Close()*: Releases the RecordSet's resources. If you have multiple references to the same RecordSet, all references to that RecordSet are closed unless you used the *Clone()* method to obtain the reference.
- *Delete()*: Deletes records from the RecordSet.
- *MoveFirst()*: Same as **GO TOP**.
- *MoveLast()*: Same as **GO BOTTOM**.
- *MoveNext()*: Same as **SKIP**.
- *MovePrevious()*: Same as **SKIP -1**.
- *Open()*: This is the most powerful and verstatile method for retrieving data from the database. The *ActiveConnection*, *Source*, *LockType*. and *CursorType* properties of the RecordSet object can be set prior to invoking its *Open()* method to populate the RecordSet. Alternatively, this can be accomplished simply by supplying this data as parameters passed to the method. The parameters accepted by the *Open()* method are:
  - *Source:* Can be either a query to execute or an object reference to the Command object that will execture the query. When this

parameter references a Command object, the *ActiveConnection* property of the RecordSet object should be left blank. The *ActiveConnection* property should be set on the Command object instead.

- *ActiveConnection:* Either a connection string or an object reference to a Connection object.
- *CursorType:* One of the values listed in Table 7.
- *LockType:* A value that specifies whether the RecordSet is opened read-only or with optimistic or pessimistic buffering. The default value is read-only (1).
- *Options:* A combination of values that specify the command type (Table 1) and the execution options (synchronous vs asynchronous).

So, if we want to create a RecordSet that contains all the records in the Customer table, we do not need to instantiate either a Connection object or a Command object. All we need to do is to create the RecordSet and invoke its *Open()* method like this:

```
oRs = CREATEOBJECT( 'ADODB.RecordSet' )
lcstr = 'provider=vfpoledb.1; data source=' + ;
        HOME(2) + 'data\testdata.dbc'
oRs.Open( 'SELECT * FROM Customer', lcStr )
```

- *Requery()*: Very similar to the **REQUERY()** function in Visual FoxPro. Used to execute the same parameterized query after changing the value of the Parameter object.
- *Resync()*: Similar to using **REFRESH()** to refresh the contents of a Visual FoxPro view.
- *Save()*: Save the contents of a RecordSet to a file. It accepts two parameters:
  - *Destination:* The name of the destination file.
  - *PersistFormat:* As of ADO 2.1 and later, this can be adPersistXML, which has a value of 1, to save the RecordSet as XML.
- *Update()*: Commits changes to the current record. Like the *Addnew()* method, the values of the Fields collection may be modified and updated at the same time by sending two arrays, one containing the fields and

one containing the values, as parameters to this method. Alternatively, the values of the fields may be modified individually by setting their values and the *Update()* method called afterward to commit the changes.

How do I use the RecordSet's Fields collection?

The *Fields* collection of the RecordSet object has methods of its own that are worth knowing something about. In addition to methods of the collection itself, the Field object has properties and methods that we need to understand in order to work effectively with ADO RecordSets.

The *Fields* collection has the following methods:

- *Append()*: Adds a new field to the collection. This method is used to create a RecordSet without using a database. It takes the following parameters:
  - *Name:* The name of the new field.
  - *Type:* A byte value from the DataTypeEnum (**Table 9** contains a partial list) that specifies the data type of the new field.

***Table 9**. Partial list of DataTypeEnum values.*

Constant	Value	Visual FoxPro data type
adChar	129	Character or Memo
adBoolean	11	Logical
AdDbDate	133	Date
AdDbTime	134	DateTime
AdSingle	4	Numeric
AdDouble	5	Double or Float
AdInteger	3	Integer
AdCurrency	6	Currency

  - *DefinedSize:* Field width.
  - *Attributes:* A combination of values from the FieldAttributeEnum used to defined attributes on the field; for example, whether or not it may be null.
  - *Value:* The value with which to populate the new field.
- *Delete()*: Removes a Field object from the collection. It takes as a parameters either the index of the field or its name.

The Field object has a number of its own properties, many of which have already been mentioned in the explanation of the *Fields* collection *Append()* method. Each of the arguments

accepted by this method represents a property of the Field object. The Field object has other properties as well. The important ones (other than those listed previously) are:

- *ActualSize*: The same as `LEN( ALLTRIM( Field ) )` in Visual FoxPro.
- *NumericScale*: The number of decimal places to the right of the decimal point.
- *OriginalValue*: The same as `OLDVAL()` in Visual FoxPro.
- *Precision*: The maximum number of digits, including those to the right of the decimal point, that the field can hold.
- *UnderlyingValue*: The same as `CURVAL()` in Visual FoxPro.

The Field object also has a couple of methods that allow you work with large strings and binary data types. *AppendChunk()* is used to place data into a Blob field. *GetChunk()* retrieves data from a field and returns it as a variant.

How do I convert a cursor into an ADO RecordSet? (*Example: CH17.vcx::xFormatter*)

As we have demonstrated in the preceding sections, this is a very simple task indeed if you are using Visual FoxPro 7. You can use the OLE DB provider for Visual FoxPro 7 to create a RecordSet using the Command, Connection, or RecordSet objects. However, if you are not using Visual FoxPro 7 (and if you aren't, why not?), it's not so easy. Or is it? We have provided a formatter class with the sample code for this chapter that has a method called *Cursor2ADO()* that you can use to build an ADO RecordSet manually from the specified cursor. Using it is simple.

The first thing that is required is to open the cursor and instantiate the formatter before calling its *Cursor2ADO()* method like this:

```
USE ( HOME( 2 ) + 'Customer' )
oFormatter = NEWOBJECT( 'xFormatter', 'CH17.vcx' )
oRS = oFormatter.Cursor2ADO( 'Customer' )
```

How does the Cursor2ADO method work?

This method creates an ADO RecordSet object and, after setting the required properties, loops though the fields in the cursor, manually adding Field objects to the Recordset's *Fields* collection.

```
lnFieldCnt = AFIELDS( laFields, tcCursor )
loRS = CREATEOBJECT( 'adodb.recordset' )
WITH loRS
  .CursorLocation = ADUSECLIENT
  .LockType = ADLOCKOPTIMISTIC

  *** Loop through the lafields array and add the field to the recordset
  FOR lnFld = 1 TO lnFieldCnt
```

The data type of the Visual FoxPro field is converted to the required enum value by the class's *DataType2ADOConstant()* function. The field attribute enum values are obtained and the *Fields* collection's *Append()* method is invoked.

```
    lnDataType = This.DataType2ADOconstant( laFields[ lnFld, 2 ] )

    *** If we had a general field, we don't want it in the recordset
    IF NOT ISNULL( lnDataType )

      *** Get the field length or default it for memo fields
      lnLength = IIF( laFields[ lnFld, 2 ] # 'M', laFields[ lnFld, 3 ], 256 )

      *** Set the field attributes (for example,
      *** whether or not nulls are allowed)
      lnFieldAttributes = IIF ( laFields[ lnFld, 2 ] = 'M', ;
        ADFLDLONG + ADFLDISNULLABLE, ;
        IIF( laFields[ lnFld,5 ], ADFLDFIXED + ADFLDISNULLABLE, ;
          ADFLDFIXED ) )

      *** OK, add the field to the recordset
      .Fields.Append( ALLTRIM( laFields[ lnFld, 1 ] ), ;
                      lnDataType, lnLength, lnFieldAttributes )

    ENDIF
  ENDFOR
```

After all of the Field objects have been added to the RecordSet's *Fields* collection, the RecordSet is opened and the fields are populated with data by scanning the cursor. A new record is added to the RecordSet for each record in the cursor by invoking the RecordSet's *AddNew()* method. Then the values of the Field objects in its *Fields* collection are obtained from the current record in the cursor.

```
  .Open()
  lnSelect = SELECT()
  SELECT ( tcCursor )
  SCAN
    .AddNew()
    FOR lnFld = 1 TO lnFieldCnt
      IF laFields[ lnFld, 2 ] # 'G'
        lcField = ALLTRIM( laFields[ lnFld, 1 ] )
        luValue = EVAL( lcField )

        *** Do not even populate the field if this is a date and it is empty
        *** since many apps cannot handle empty dates
        IF NOT EMPTY( luValue )
          .Fields( lcField ).Value = luValue
        ELSE
          IF NOT INLIST( VARTYPE( luValue ), 'T', 'D' )
            .Fields( lcField ).Value = luValue
          ENDIF
        ENDIF
      ENDIF
    ENDFOR
  ENDSCAN
ENDWITH
```

How do I convert an ADO RecordSet into a cursor? (*Example: CH17.vcx::xFormatter and TestADO2Cursor.prg*)

If our Visual FoxPro application must accept data in the form of an ADO RecordSet, it makes good sense to convert it to a native cursor before processing it—unless, of course, your application happens to be running a little too quickly and you would like to slow it down a bit. The exposed *ADO2Cursor()* method of our *Formatter* class handles this job with the greatest of ease. It does not matter that there is no native function to convert an ADO RecordSet into a cursor because our generic class does what is required.

The method accepts two parameters. The first is an object reference to the ADO RecordSet that requires conversion. The second is the name of a cursor to hold the results. The first thing that the method does is to construct an array from the *Fields* collection of the RecordSet, and it uses that array to create the result cursor. The class's custom *ADOConstant2DataType()* is used to convert the Field object's *Type* property to a Visual FoxPro data type.

```
lnFld = 0
FOR EACH loField IN toRS.Fields
  WITH loField
    lcType = This.ADOConstant2DataType( .Type, .DefinedSize )
    IF NOT ISNULL( lcType )
      lnFieldSize = .DefinedSize
      lnPrecision = 0

      *** Now, get the precision if this is a numeric type field
      *** and set the field size for memo, datetime and logical fields
      DO CASE
        CASE lcType = 'L'
          lnFieldSize = 1
        CASE lcType = 'M'
          lnFieldSize = 4
        CASE lcType = 'T'
          lnFieldSize = 8
        CASE INLIST( lcType, 'I', 'N', 'B', 'F', 'Y' )
          lnPrecision = .Precision
      ENDCASE
      lnFld = lnFld + 1
      DIMENSION laFields[ lnFld, 5 ]
      laFields[ lnFld, 1 ] = .Name
      laFields[ lnFld, 2 ] = lcType
      laFields[ lnFld, 3 ] = lnFieldSize
      laFields[ lnFld, 4 ] = lnPrecision
      laFields[ lnFld, 5 ] = BITTEST( .Attributes, 5 )
    ENDIF
  ENDWITH
ENDFOR

*** Go ahead and build the cursor
CREATE CURSOR ( tcCursor ) FROM ARRAY laFields
```

Once the cursor is created, all that is left to do is to iterate through the RecordSet and transfer its data to the cursor.

```
toRS.MoveFirst()
DO WHILE NOT toRS.EOF
  APPEND BLANK IN ( tcCursor )
  FOR EACH loField IN toRS.Fields
    WITH loField
      *** Make sure it corresponds to a field in the cursor
      lcFieldName = .Name
      IF TYPE( tcCursor + '.' + lcFieldName ) # 'U'
        REPLACE ( lcFieldName ) WITH .Value IN ( tcCursor )
      ENDIF
    ENDWITH
  ENDFOR
  toRS.MoveNext()
ENDDO
toRS.Close()
```

Conclusion

ADO RecordSets can be a great way to transfer data back and forth between Visual FoxPro and Automation servers such as Word and Excel. Although the newer versions accept XML, the older ones do not. Sending the data to these applications as ADO RecordSets eliminates the need for your Visual FoxPro applications to have intimate detailed knowledge of the inner workings of any document, template, or spreadsheet. It allows you to take a “throw the ball over the wall” approach, just sending the other application the required data. It now becomes the VBA programmer’s responsibility to populate the Automation server’s document. As a Visual FoxPro developer, the only thing that you need to know is the public interface: how to send the data to the server and how to ask for any results, if there are any.

Chapter 18
Testing and Debugging

The key to a successful acceptance of your application by your customers is to make sure that it meets the documented requirements. The only way to prove that it meets the documented requirements is to test each piece to see if it conforms to their needs. This chapter will first discuss testing, the types of testing that you can perform to prove the application worthy of being deployed, and techniques we use to test various components of an application. Once defects are discovered during testing, we need to jump into debugging the problems discovered. We discuss the scientific approach to debugging, and specific debugging tips in the Visual FoxPro debugger.

What exactly is testing? Simply stated, it is the verification that the developed software meets or exceeds the functionality defined in the specifications. If there is one guarantee we can make about software development, it is that if you write code, at some point you will write defective code. Testing to validate the specifications has the assumption that a specification was developed. It is important that the customer reads the specification and agrees with your view of what they need. The cost of finding defects in software gets higher the further we get into the software development cycle. Meaning, it is cheaper to fix a defect when reviewing the specification than it is during the construction phase, and it's definitely cheaper than fixing it after a deployment. The cheapest time to repair the defect is at the moment it is discovered. Sounds obvious, right? The difficult part is finding the defect and determining if something really is a defect.

Common sense dictates that there are classes of software failures that defects commonly fall into:

- Improperly constrained input
- Improperly constrained output
- Improperly stored data
- Improper computation
- Usability
- Platform inconsistency
- Improper documentation

Be ready to test all cases. This author will never forget one bet he made with his manager for a lunch. The manager challenged me that he could break the application in less than five minutes. Developers are a proud bunch, and I was no different. I was very confident with this simple FoxPro DOS application and the quality I had put into it. So when I was ready I called him over to make good on the lunch he was about to buy me. He spent about four-and-a-half minutes executing different features, adding records, deleting records, editing data, seeing if referential integrity code fired, altering data to extreme limits, making sure reports printed and

could be previewed, and verifying that data was correct. I was very sure that the free lunch was secured. At that moment he looked up at me and said, "Just one more thing," and then took both hands and crushed them down on the keyboard. The program locked up hard. My jaw hit the desk. I argued that no user would ever do that. He noted that the average user of the application used to register people coming in the door of a secure facility was a retired fireman, and that he could easily fall asleep on the job and his face could hit the keyboard in a similar fashion. I added this to the requirements documentation, added a `SET TYPEAHEAD TO 0` in the code, and took him out to lunch.

This example was a simple defect to find and fix. Our industry has a number of well known testing failures. The horribly buggy releases of dBASE IV, MS DOS 4.0, Netscape Navigator 6.0, and the Pentium math-coprocessor overflow defect might have been avoided with better testing processes and release decisions. These were all made by industry leading companies and lead to some embarrassing times. While the magnitude of defects we release might not be worldwide and cost millions of dollars to repair the public relationship, we do not want our customers to be impacted to the point that they do not trust us when it comes time to deploy the next release.

The first part of this chapter will address the testing process. You will see how to determine when an application is ready for release, and outline the different types of testing that your software development shop and your customers can perform. We will show the different things to include in a test plan and things to consider when preparing different types of releases. Testing techniques will be discussed as well as some Visual FoxPro tools (both automated and manual) to assist in deploying code that is as defect-free as possible. We spend some time on code walkthroughs and show how it is an excellent way to test components of an application.

The second part of this chapter will address the science of debugging the defects found in testing. You will see how using a deliberate approach like the scientific method is better than the shotgun approach to debugging, and why it is important. We conclude by showing a number of Visual FoxPro debugger tips and some improvements in the debugging capabilities of Visual FoxPro 7, and provide a couple of handy developer tools as well.

How do I know an application is ready?

There are three perspectives of when the product is complete and ready for deployment: the engineering viewpoint, the quality assurance check, and the customer determination that it meets requirements. All three viewpoints will have an impact on the decision for a release go, no go, based on a predefined list of acceptance criteria.

The engineering viewpoint is comprised of input from the project leader (the customer's representative to the development staff), the project's developers, and other developers in the company or a quality assurance team. In a smaller shop, one person may be playing all the roles. It should be obvious that the product is not ready until all the features have proven to meet the customer's stated requirements. This is unit tested by the original developer, and system tested by the other developers and the project leader. This proves that the interface supports the business rules, which are implemented in data, and that they all work together to manage the information correctly. Further, all company guidelines and standards have been implemented in the code that was developed to support the solution that is ready to ship. These guidelines include proper coding standards, proper implementation of frameworks, proper use

of third-party tools and controls, and proper implementation of industry standards if appropriate. The way to enforce these standards and guidelines is to either have a team walkthrough or a simple desk check by one other developer. We have more specifics on this process later in this chapter. The goal of these review sessions is to make sure the code is matching requirements, is readable, is supportable, and is understandable. A one-person shop has to do it all, but even one-person shops typically have contacts they can call on for review of code if needed. If not, take some time to print code off and review it when you are away from the computer.

The quality assurance perspective is almost self-explanatory. A quality assurance team strictly enforces that the customer's requirements are implemented correctly. We think the real issue is how many software shops really have a staff dedicated to quality assurance. We are not talking about 100 people dedicated to the testing of all developed products. This could be one person who knows how to be the "unknowledgeable user" and the "knowledgeable user" at the same time. This role can be filled by another developer(s), and is best filled by people who were not involved from a coding aspect. A key to success with a QA department/person is to have them involved from the start of the project. By having them review the functional specification and prototype, they can start building the test cases for the test plan.

The customer's acceptance testing is critical; otherwise, you likely will not be paid and it will be time to concentrate on another hobby that can be turned into a paying job. On the other hand, do not rely on the customers to find your defects. It is important to note that the customers are usually not trained in finding defects in software. The best you can hope for is that they are business experts and will find all the process defects and miscalculations. They will likely point out every flaw in the interface (labels misspelled and unaligned by a pixel on a form). We always suggest you watch them struggle to add a new thingamajig into your latest software creation. See how they use it or, more importantly, how they fail to use it. They may not even point out missing features for months. A thing like end-of-the-month processing does not get tested until the end of the first production month. Even when you step them through the process during "testing," seldom-used features typically get their attention months down the road. The key to a successful user acceptance test is to give them a test plan when delivering a test version. Make sure every requirement is somehow tested by the test plan. We discuss test plans later in this chapter.

What types of testing can be performed?

There are several recognized tests that can be performed on software. It is important to recognize who performs the testing, how it is done, when to perform the testing, the benefits, and what the expected outcomes are of the testing.

Testing is performed in stages. Some developers test their code as they develop it, others test it after it is all developed, and others wait for customers to test. We strongly believe that code should be tested in four iterative stages: unit, integration, system, and user acceptance (see **Figure 1**).

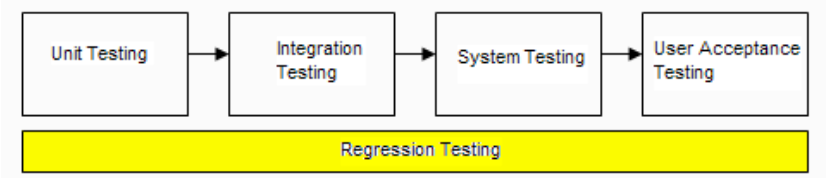


***Figure 1**. The stages of testing various aspects of an application are performed in a sequence from unit testing through user acceptance testing.*

At any point in the testing process a defect may be discovered. A defect could be in the form of a non-compliance with a requirement, or the discovery of a new requirement. These defects should be recorded and it should be determined if this is a show-stopper, an issue to be addressed before release, an issue to be addressed after release, or an issue never to be addressed. If an issue is addressed, the component will start the test cycle all over again. All stages can optionally have regression testing.

Unit testing

Unit testing is performed by the developer to test individual software components or a collection of components. A component could be a program, a class, a set of classes, a form, a menu option, or a report. Developers define the input domain for the module in question and ignore the rest of the system. Unit testing sometimes requires the construction of throw-away driver code, self-testing methods, stubs, and is often performed with the use of a debugger.

Unit testing is performed during the construction phase of development. Developers test each unit as they write the code. It is an iterative process. Write code, execute, see if it works, fix if it is broken, and then write more code. The benefit is that the code is tested immediately as it is written, by the developer most intimately aware of the requirements. The expected outcome is that the unit is tested and verified to meet the requirements within the confines of the unit.

To demonstrate unit testing we are going to show how it can be done with a business object. A business object is developed to enforce several business rules. In the example we will discuss a business object that enforces business rules for a customer—business rules like all new customers added are assigned a customer ID, and require a full address, phone number, and contact person. The developer might develop a custom object that has methods like *AssignId()*, *ValidateAddress()*, *ValidatePhone()*, and *ValidateContact()*. Properties might be added for the address, phone, and contact. One additional method is added called *SelfTest()*. The *SelfTest()* method contains code needed to internally test out the methods and properties of the business object.

```
LPARAMETERS tnTest

IF VARTYPE(tnTest) # "N"
  MESSAGEBOX("Invalid parameter type passed to " + LOWER(PROGRAM()) +;
             " method." + CHR(13) + ;
             "It should follow " + LOWER(PROGRAM()) + "(tnTest) syntax.", ;
             0 + 16, _screen.Caption)
  RETURN .F.
ENDIF
```

```
DO CASE
  CASE tnTest = 1
    * Data provided
    WITH THIS
      .cAddress    = "1234 Main Street"
      .cCity       = "Anywhere"
      .cRegion     = "MI"
      .cPostalCode = "48000"
      .cPhone      = "800.555.1212"
      .cContact    = "Ms. Leader"
    ENDWITH

  CASE tnTest = 2
    * All data fails
    WITH THIS
      .cAddress    = SPACE(0)
      .cCity       = SPACE(0)
      .cRegion     = SPACE(0)
      .cPostalCode = SPACE(0)
      .cPhone      = SPACE(0)
      .cContact    = SPACE(0)
    ENDWITH

  OTHERWISE
    lcOldAssert = SET("Asserts")
    SET ASSERTS ON
    ASSERT .F. MESSAGE "Invalid parameter passed to " + ;
                       LOWER(PROGRAM()) + " method."
    SET ASSERTS &lcOldAssert
ENDCASE

WAIT WINDOW "Id Assigned = " + TRANSFORM(.AssignId())
WAIT WINDOW "Address result = " + TRANSFORM(.ValidateAddress())
WAIT WINDOW "Phone result = " + TRANSFORM(.ValidatePhone())
WAIT WINDOW "Contact result = " + TRANSFORM(.ValidateContact())

RETURN
```

To complete the unit testing, the developer will instantiate the business object and call the *SelfTest()* method for each case that needs to be tested. The developer can still set properties and call methods as well.

Obviously, testing a user interface object requires interacting with the object. A class might require you to add it to a form and interact with it. A program or report would require actually running the appropriate code and verifying the results. The key is to test all the functionality. There are other sections later in this chapter that specify some of the things we test for when unit testing.

The benefit of this testing is that each component is verified individually and certified to be operational by the developer. Once the developer certifies that the unit is working, it moves into integration testing.

Integration testing

Integration testing tests multiple components that have each received prior and separate unit testing. In general, the focus is on the subset of the application that represents communication between the components. This testing is performed to ensure that components that were tested

individually can collaborate together as specified. Integration testing is conducted after the developer finishes unit testing. Many developers will perform integration testing immediately after unit testing to further validate the unit testing.

Like unit testing, integration testing is performed during the construction phase and is handled by the developers who wrote the components, or by other developers on staff. The way it is accomplished will depend on the components to be tested. One example of integration testing might be a query object and a report. The query object might be responsible for presenting the user interface, generating the query based on the user selection, and generating the resulting cursor. The report is developed separately. They must work together to generate the report output. Another object might be developed to allow the user to select the output type (printer, screen/preview, HTML, PDF, and so forth). All three components need to be tested as one unit to verify that it is working to the specifications.

This testing can be performed with the components in the Visual FoxPro development environment or compiled into the deployable format (APP/EXE/DLL). The expected outcome is a verification that all the components work together properly. If they do not, the failures need to be documented, fixed, and retested. The units that are fixed need to be unit tested and returned to integration testing.

At this point in the testing the developers will have proven that components are working together. Other type of testing completed at this stage include making sure that it is functionally correct, computationally correct, and that all exception cases are verified.

Functionally correct

Functional testing requires the selection of test scenarios without regard to source code structure. Test selection methods and test data adequacy criteria must be based on attributes of the specification or operational environment and not on attributes of the code or data structures. Functional testing is also called specification-based testing, behavioral testing, and black-box testing.

Computationally correct

Computation testing ensures that calculations performed in the application are performed correctly and to the specification. This testing often requires that data be set up in advance so that reports can be run and both detail and summary calculations verified. The same setup/testing needs to be accomplished with forms that have calculations and summary information presented.

Exception testing

Exception testing is performed during all testing up to integration. It makes sure that upper and lower boundaries are not exceeded, and that all business rules are enforced correctly. This is the type of testing that makes sure you cannot enter negative numbers where inappropriate, that the asterisks do not show up on the reports because of numeric overflow, that field sizes are not exceeded because of data entry, and that fields that do not accept nulls are not exposed in the user interface to accept them. Each of the known and documented business rules should be tested as well.

System testing

System testing validates the requirements for a collection of components that constitutes a deliverable product by executing the system with the intent of finding errors. The entire application must be considered for testing to satisfy a system test. The system test is completed after the various components pass the integration testing. System testing is performed after integration testing and before user acceptance testing. The purpose of system testing is to compare the system to some system specification defining the product.

System testing involves really "working" the application. The goal here is to be sure that you can access and run features together, and to find contradictions to the specifications so errors are detected and corrected (or managed) prior to presenting the system to the customer for user acceptance testing—that numerous forms can be opened, that modal functionality is indeed modal, that adding/changing data in one section of the application is reflected in another, that security works if it is integrated in the application, that reports reflect correct information, that validation is working as expected.

This is the first testing we have discussed in this chapter that can involve the customers, but is usually performed by developers. The development staff is most knowledgeable about the application and should have a test plan developed that consists of the various interactions with the application that need to be tested. If the customers are involved, the developers should be working side-by-side with the customers. Hopefully they will be stepping the customer through the test plan document. They also can be training them on how to use new features of the application and reminding them to test existing functionality if applicable. The key to a successful test is to make sure you can open each of the components, exercise the functionality, and validate the requirements.

One of the biggest benefits of system testing is cross-training developers for support. Developers get to learn about the application as they test. At this point in the development cycle developers who have not worked on developing the components of the application can learn about the application. If they have developed components of the application they can learn about other parts of the product by testing those features.

The expected outcome of the system test is that the application is thoroughly tested and meets all expectations stated in the requirements documentation. At this point the development staff needs to determine if the application needs to be changed because failures were uncovered, or if the product is ready to be tested by the customer.

System testing includes a number of "specialized" approaches to testing that can be performed. These include usability, load and performance, conversion, recovery, installation, and platform testing.

Usability

It might sound strange that you would be testing the usability of the application as late as system testing. We are not suggesting that this is the first time this is tested. Usability is something that should be considered at all phases of development, especially starting with the design, long before the first line of code is developed. This usability testing is in the context of the entire application. Can you bring up multiple forms, does the form manager handle the cascading of forms correctly, can you edit data in various forms, and can you approach changes in multiple forms at the same time? All human factors of using this application should be verified one last time.

Load and performance

Load testing ensures that the components and the system work with empty datasets and a data set that is populated with a number of records that matches or exceeds the expected maximum number of records to be maintained by the application. This testing helps avoid those embarrassing moments after you deliver the application for the first time and the user goes to add the first record and it does not work because the empty test was not completed. It is a great way to test the referential integrity rules enforced by the database. The populated database will test performance of the application, especially when it comes to verifying that Rushmore optimization is implemented. It will also ensure that the performance of client/server queries does not affect performance either.

Conversions and data imports

Conversions can take place in two formats. The first is preloading data from a previous application into the current one. This happens during application rewrites. This could also be complicated when moving data between different operating systems, different database platforms, or from different data structures (imports). The other is converting changes to the data model between releases of the same application. Adding or dropping columns from a table, possibly filling in data to the new columns, adding or dropping tables or views, changing the data type of a column, or adding stored procedures are possible conversion issues that need verification.

Recovery

Recovery testing is the verification that the application can recover from power outages, hardware failures, or application errors. We have seen developers get very nervous when testing their applications and in the middle of a batch process shut off the computer. What happens when the program terminates based on an error that was not handled specifically? It is important to make sure the error and state of the application gets recorded properly.

Installation

Make sure all installation routines are tested out. This verifies that the runtimes are loaded to the appropriate location, the data is loaded and updated, the EXE is loaded, the ActiveX controls are installed, and the Registry entries are made correctly. Testing this on several machines with different configurations might be necessary depending on the expected customer base.

Platform

Platform testing ensures that your application runs on the various operating systems (Win95/98/Me/NT/2000Workstation/2000Server/XP/Novell). Software can also fail by not satisfying environmental constraints that fall outside the specification—for example, if the code takes too much memory, or executes too slowly, or if the product works on one operating system but not another. These are considered failures.

Testing on each of the various platforms can be time-consuming and expensive. This is why it is important to determine the minimum configuration that your application will require for support. While it is optimal to perform this testing during development, often this type of testing is performed near the end of the development cycle. The majority of the cost involved is configuring the hardware to perform this testing. With tools like Norton's Ghost and

PowerQuest's DriveImage we can configure a computer with the operating system and necessary drivers and save an image of the partition where the operating system resides. These images can be used later to restore the machine to the various configurations necessary for testing. We also recommend using older machines that have been retired from day-to-day development for the purpose of testing. These machines can be configured with the platforms supported.

This testing is definitely handled by the development staff and the quality assurance team. They will make sure that all Windows API calls work, that interaction with devices like printer drivers, COM ports, networking hardware, video drivers, memory configurations, modems, fax hardware, and audio devices all meet expectations.

The expected outcome is a list of operating systems and configurations that are supported for the application. Another possible side effect of this testing is that developers might need to evaluate changes to the application to make it work on a platform that it failed to perform correctly. If the application needs a certain platform to run and the customers do not have hardware to support it, it is better that they are informed of this fact as early as possible so they can allocate funds to acquire the new machines, or decide to cut losses and not move forward with the software development or purchase.

Another approach to perform platform testing is discussed later in this chapter. See the section on "How can I test apps on various platforms without reloading the OS?"

User acceptance testing

User Acceptance Testing (UAT) is when the users of the component or application determine whether it meets/exceeds their requirements. This testing is naturally performed by the users, with assistance from the developers. The assistance can be in the form of a test plan, sitting next to the users as they test, or a combination of the two.

User acceptance testing is commonly performed at the end of a system life cycle. UAT can be performed at the component level or system level. Each component can go through the cycle of unit, integration, system, and user acceptance testing. This is an important point since users should not see the application for the first time just before it is implemented.

The expected outcome of UAT is that the component is approved for release by the users. If the UAT is being performed at the application level it is either ready to move into production, or needs some changes. At this point the documented failures need to be evaluated. If the problem is a serious defect, the defect needs to be returned to the developers to be fixed. If the problem is a change in requirements, a change order or other mechanism needs to be completed to document the new functionality. Prioritizations of the changes need to be made. It might be that the component or application is implemented as is and that the changes are made later. It also might be determined that the changes need to be made before the implementation.

Regression testing

Regression testing is performed when developers create a new version of the software in which a reported defect has supposedly been removed or a new feature added. The question is, how much retesting of the new version (n) is necessary using the tests that were run against the old version (n – 1)?

Any specific fix can fix only the problem that was reported, fail to fix the problem, fix the problem but break something that was previously working, or fail to fix the problem and break something else. Given all these possibilities, it would seem prudent to rerun every test from prior versions on the updated version before testing anything new. The practice of regression testing is generally considered cost-prohibitive. Moreover, new software versions often feature extensive new functionality, in addition to the defect fixes, so the regression tests take time away from testing features. To save resources, testers need to work closely with developers to prioritize and minimize regression tests. One major drawback to regression testing is that new features often supersede or alter the functionality to where the regression tests "fail." Developers and testers need to work together to eliminate tests that are no longer valid.

Regression testing is performed by the developers during unit, integration, and system testing. Users also perform regression testing during user acceptance testing. The benefits are obvious; the previously implemented features are retested to make sure they still work. There are many ways for developers to embarrass themselves in front of the customers. Why risk breaking something they have used successfully in the previous release? The expected outcome is that the functionality works as it did in the past, unless the new fix or enhancement altered the behavior.

Skipping regression testing is always an option. It is a matter of managing how much risk is involved and how much potential there is to breaking another feature. The level of intensity and the amount of regression testing can be determined on a case-by-case basis, but we recommend as much regression testing as is practical for the release. Obviously, the more regression testing performed, the less likely you are to introduce new defects by correcting one.

What is a test plan?

A software test plan is a written document that describes the objectives, scope, and methodical approach of a software testing process. The process of preparing a test plan is a useful way to think through the efforts needed to validate that the requirements of the application are met. The completed document will help people besides the developer understand how an application is tested to the requirements. It should be thorough enough to be useful, but not so thorough that no one will read it or use it.

The objective of a test plan is to provide a plan for the verification of the requirements. By testing the software, a person following the test plan ensures that the software produced meets or exceeds the functional specifications. The objective can be simple as: Verify the requirements of application X and that they meet the release criteria as documented and agreed upon by our company and the customer. The identification, tracking, and reporting of discrepancies discovered during testing is also specified.

A test plan states what items are to be tested, the level at which they will be tested, and the sequence in which they are to be tested, and describes the environment necessary to complete the test. The scope should be the first consideration when preparing the test plan. Who is the intended audience? Is the audience the development staff, the test team, or the customer? Is it unit testing, system testing, or acceptance testing? Changes to the audience and type of testing will dictate what is included, how much effort is expended, and completeness.

What should be included in a test plan? It is important to note at this point that what is included in the test plan will depend on a number of circumstances, including but not limited

to the objective, the scope, the type of testing conducted, who conducts the testing, the extent of the application, and what type of release (defect fix, brand-new software, general upgrade).

All test plans should have an introduction, a stated purpose, and a list of objectives. This sets the tone for the people conducting the testing. It clarifies exactly what we are attempting to accomplish. This does not have to be elaborate; in fact, the simpler the better because it is easy to understand clear and concise objectives. If necessary, include a short overview of the application, and include any references like specification documents and data models.

Configuration of the test environment is important. This is where you define what equipment is necessary (PCs, peripherals, networking), the test data (datasets that test various test conditions), where the application resides, and any necessary deployment concerns you might have in configuring the test. It is also helpful to determine the type of equipment that is necessary for the users to have in place and the minimum configurations necessary to run the application. If you need to load test data into a directory or SQL Server, you will want to plan so the data is prepared and can be loaded when testing starts. Outlining the data needs will also help prepare the test data in advance (conversion from old system, load the test cases, and so on).

Depending on the organizations involved, it might be a good idea to document the people resources and gather an outline of their responsibilities in the test plan. This helps the test team understand exactly what is expected of them as the test plan is executed.

Highlighting the risks and dependencies is something to be considered. If you know that the customer is not particularly dedicated to testing, or they cannot test the application at the end of the month because they are busy meeting widget-building deadlines at that time, then note it as a risk factor. Fitting in testing can be a difficult task. You may find that generating a schedule for the test might be necessary. If you are testing out a weekly batch process and a monthly batch process, you may have to coordinate this with real timeframes, and schedule the testing in a certain sequence.

Document the testing strategy. How do you expect the testers to proceed through the test plan and test cases? Is the tester supposed to test only the documented test cases and not make up their own? If they come up with a new test case, what is the process to getting the test case added to the existing list of test cases? Can a test case of "general form testing" be included, and does the tester know to go through the common form-testing checklist?

The meat of the test plan will be a list of features to be tested, the individual test cases, the test data to be used, and an expected result for each test case. Each test case should have pass/fail criteria and room for the tester to indicate the pass or fail. It is a good idea to leave room for the tester to comment on the test.

Just as important as the list of features included is a list of features not to be tested. There's no sense in wasting the tester's time in testing a feature that is not ready for testing. This is self-explanatory.

Include in the document the results forms. Each test will result in a grade of pass or fail. If the test failed, it needs to be classified as a show-stopper (data corruption), an issue that needs correction (a process that throws a bad error message), or an issue that does not need correction at this time (a label not aligned correctly on a form or report). The release acceptance criteria will play an important role in determining what is acceptable, and what is not acceptable. The mechanism for error tracking, reporting, and classification should be outlined in the test plan.

We like to include a history of the document since it is a living document. Items are added over time depending on the change made to the application and refinement of the testing process. Noting when the changes were made, the extent of the changes, and who made them can help future developers and testers better understand how the document made it to the current state.

The all important signoff section needs to be included. There should be room for all testers to sign off on the testing performed. This implies that the testers are dedicated and in part assume ownership in their testing and contribution to the project.

When should you write the test plan? It is easiest to start as you are writing the specifications. This is not always practical, so the next best time is during the development. The idea here is the sooner the better. Leaving it to the end of the construction phase is too late. At this point you will want to hand off the testing to someone other than the original developer (another developer, dedicated testing person, or customer). Waiting until this point will surely delay effective testing because the person writing the test plan will need to spend time doing so.

Who should write the test plan? If you are a one-person shop, you are likely to write it, but the test plan can be written by anyone who understands the requirements. This can be developers, it can be quality assurance team members, and it can be a customer who is technically proficient enough to do so.

So where can you find the test plan example in the chapter downloads? There is none. It is not important how the document is formatted, it is important that it is functional for your organization. Assembling a test plan is a matter of firing up your word processor and writing up the items outlined in this section as you see fit. There is no way we can cover a complete test plan within the confines of one section of one chapter of a book on extending Visual FoxPro. There are numerous sites on the Web that are dedicated to test plans and quality assurance in general. There are conferences dedicated to the subject, numerous books, and white papers as well.

How do I test various types of releases?

Is there a difference testing when shipping different types of applications? Is there a difference between shipping a bug fix, a complete new system, or an upgrade when it comes to testing? What is the difference between delivering standalone components and a standalone app? Nothing, other than the amount of testing physically required and determining the amount of risk involved with deploying potential defects discovered and not discovered.

The same intensity used to system test a new app should be used to verify that the latest defect found by the customer was squashed. The key is to test everything that is affected by the development completed. Maybe the engineering testing calls for you to desk check a defect fix and walkthrough a major change, but all of it gets tested by someone other than the developer and the customer before it is declared ready for implementation. It should be tested in a test environment that mirrors a production environment to make sure it works for all cases, not just the case set up with the developer's test data.

We always use the built-in EXE version numbers to differentiate between builds that go to the customer. This way we know when the customer asks why a feature is not working in the version they are using (2.1.235) that the answer is because it was added in the current version and that they need an upgrade (to 2.3.309).

How do I manage the risk of releasing defects?

It is important to remember that shipping is a feature. So how can you manage the risks of releasing known defects in the application to the users? Determine what the impact of the defect is on the users and evaluate whether it is worse not to deliver the rest of the working application. It may not be important to hold the release because one subtotal on a report is not correctly counting the number of widgets when they need the new interest calculation on the invoices to increase their revenues. There are a number of factors that will weigh in on this decision.

Is the application a vertical market app or a custom application for one customer? Vertical market application releases can sometimes impact a larger customer base than a custom application designed for one department of a small company. Releasing a defect will translate into education of the customer base, higher deployment costs (when releasing follow-up fixes), and a higher cost for technical support. Deploying a problem to one site is cheaper than deploying to 100 sites.

Is the defect occurring in a frequently used or mission-critical feature? The release will likely be delayed for defects in frequently used features more than a feature used infrequently or one that is not impacting a business as much.

Allow your customers to influence the go, no go decisions by co-developing the release criteria. This can be done by prioritizing the feature sets and list of test cases. Go down this list one-by-one and determine whether the failure of the test will be considered a show-stopper. During the testing, if the failure occurs, get it fixed or delay the release. Once the testing is completed, review the release criteria with the customer and make sure they are still comfortable with the results and decide if the release is a go. If defects occurred and the release is still a go, make sure the known issues are documented and presented to the user base upon installation. While there is no guarantee that the users will read the problem list, at least you have attempted the proper disclosure and can refine your techniques in the future to better inform customers of potential pitfalls.

How can I test forms?

There are some straightforward steps to test that a form is working correctly. Not all of these items will apply to all forms. These steps are not taking into account testing the business logic in the specification; these are items to test that are generic in nature. Making sure that the list is covered for each form in the application has saved us from looking unprofessional.

Verify that the form's tab order is correct. This test is one that needs to be keyboard-centric. When we find tab order problems during testing it is usually a developer who is a wizard with a mouse. During this testing you might want to make sure objects that should not be in the tab order are removed by setting the *TabStop* property.

Invoke the add/edit mode and modify data in data bound objects. When adding records make sure the default values are properly set. Developers have different ways to change the mode of a form from non-edited to edit. One approach is to require the users to press the Edit button that changes the objects from read-only mode to be editable. The other approach is to auto-sense that the data has been changed. This testing also verifies that the enabling and disabling of various toolbar buttons (whether on the form or in a real toolbar) are handled correctly. Always check that the referential integrity rules are also properly enforced.

Save/cancel all changes to all data bound fields. Making sure this works sounds obvious, but there can be changes to the underlying views that can break the saving of data from a form. We have run into this more than a few dozen times when another developer accidentally changed the *SendUpdates* property on a view to `.F.` and forms stopped working correctly.

Invoke the delete capability and verify that the appropriate messaging occurs asking whether the user wants to continue with this destructive operation. If there are associated referential integrity rules in effect, make sure these are enforced.

Use different logins to validate security for the form if applicable. Security can be implemented at various levels. Is the menu option for the form enabled or even displayed for the various security levels? Some security implementations will display the form, but the data is read-only, and others will disable only certain objects or toggle visibility. Hopefully the form security implementation is well documented so it can be well tested. This testing is very important since users can get agitated when sensitive information is seen by the wrong people.

Verify all incremental searches work correctly. Incremental searches are very common and often depend on the appropriate indexes being created for the table or cursor. Just like the save/cancel testing, it is something that can be broken at the table level or metadata level.

Push all the command buttons on the form. Command buttons are used to invoke some functionality and this functionality needs to be tested. Make sure the functionality performs to specifications. If using toolbars that interact with the form, make sure to test all the toolbar buttons as well.

Invoke sort and filtering functionality. Many commercial and custom frameworks use metadata to drive this functionality. Make sure that the form refreshes lists and grids that reflect the new sort order or filter condition. If the natural language of the filter condition is displayed make sure it is correct. Sort orders are often indicated visually with an indicator or colorization of a header.

Invoke all child forms to make sure they are included in the application. Child forms are forms that have data related to the parent or calling form, but are accessed from the calling form, not from a menu or switchboard interface. Forms are often called through indirection and can be left out of executables. Invoking all the child forms will alert developers to include them in the application project.

Invoke all delayed instantiation objects. A technique to improve performance of the form is to delay the instantiation of objects on pages of a pageframe that are not often looked at by the user. The first time the page is activated it instantiates the interface object for that page. If there is a problem with this process it will not be obvious unless that page is accessed during testing.

Right-click on everything, and try all shortcuts on the form. Right-clicking can integrate context-sensitive Help, initiate a process, or provide a shortcut menu with additional functionality. Making sure all objects on the form that are supposed to have this functionality is natural, but verifying that objects that should not have these features don't is also important.

Verify all list objects have proper data and sorting. Testing includes validating the proper data in the list, that it is sorted in the proper order, and that columns are displayed with the correct widths so all the information can be read. This is another item that can be broken by changes to the underlying data and metadata. If the list is dependent on a view or an index order, the list can get populated incorrectly if someone makes a change to the view, index, or table.

Press the F1 key to make sure it brings up the context Help. This is not just for the form, but tab to each object and press F1 if field-level Help has been implemented. If the form has What's This Help, make sure that the appropriate text is displayed. This is a great way to also test out the content of the Help and to make sure the section is written. Often Help is developed by someone other than the developer, and this disconnect can lead to implementation of Help not working properly.

Verify all abbreviations used on the form meet industry, customer, and development standards. Inconsistent abbreviations are confusing and often lead to technical support calls. Verify ToolTips are appropriate and used only when needed. Implementing a ToolTip on every object might not be necessary. Validate that all picture-based command buttons and checkboxes have ToolTips.

Verify that there are no hotkey conflicts between objects on the form, and between objects on the form and the system menu (or form menu if you are testing a top-level form). This is actually a very common occurrence in our experience. It is also something that is often overlooked by those developers who are mouse wizards.

Verify all formatting is correct. This can be testing the object's *Format* and *InputMask* properties, as well as the general layout and format of the form. Make sure that the fonts used on the form are consistent with the other forms throughout the application. Align all the objects on the form so it looks professional. The alignment tools included in the Form and Class Designer in Visual FoxPro are excellent. There is no reason to have objects that are misaligned. Check all label captions and ToolTips for correct spelling. If your users are like our users, they spend more time picking on misaligned fields and spelling mistakes on the form than a buggy validation process. Save yourself some aggravation and catch these before the users do.

Validate that the form graphics are correct and being displayed. Icon and image files that are not included in the project must be distributed separately and you need to verify that this actually happens.

Verify all Stonefield Database Toolkit (SDT) metadata is set up clean and validated for cursors. This is not something everyone is going to have, but it is a popular tool. All the commercial frameworks are integrating with SDT. Several of them use the metadata for indexing, captions, ToolTips, filtering, and sorting.

How can I test reports and labels?

There are some straightforward steps to verify a report is working correctly. Not all of these items will apply to all reports. These steps are not taking into account testing the business logic in the specification; these are items to test that are generic in nature.

Make sure the correct data is printed on the report. This might sound like a ridiculous concept, but we have run across numerous test cycles with the customer where the data was not correct. One of the common problems is that the developer started with a Quick Report and it included a view in the report dataenvironment. Initial unit testing shows the report with the correct order and fundamental information. If a different view is used for the report, or more commonly, different SQL-Select code was used to prepare a cursor for the report, the view in the dataenvironment will still be used for the report. The view in this case needs to be removed from the report dataenvironment.

Test to make sure the data is correctly sorted and the report grouping is correctly bundling data together. One immediately obvious problem is that group headers and footers are printing for each record processed. That is a good indication that the report cursor is not sorted correctly or that the groupings are not correctly ordered in the report.

Validate report grouping functionality works as specified. Test that summary fields are calculated correctly and that the calculations are reset to the appropriate grouping. One common defect we have fixed over the years is a summation that is created for group footer and then copied down to the summary band, but still resets to zero each time a group break occurs. Adding up the numbers by hand to validate summations and counting the number of records for count calculations is important. Also, run the report with the `SUMMARY` clause for reports that have this capability exposed in the report to validate that it works. Validate that the title and summary pages are printed and have separate pages if specified.

Test that all Print When conditions function properly. Our experience has shown that developers write expressions for report Print When conditions the same as menu `SkipFor` clauses. These are opposite logic and need to be verified accordingly.

Line objects that connect correctly in the Report Designer do not always connect correctly in the preview mode or when printed depending on the printer driver and video drivers. Most of the time they are fine, but our experience has not always found success in this regard. Make sure to test this on a number of printers, especially the ones that the customer uses.

If you are testing a label, test it on the actual label it was designed to print on. There are a number of predefined labels in the Visual FoxPro templates that do not exactly match the specification of the Avery labels that they were designed to print.

Verify reports to different printers if possible. Definitely test the reports on printers of the targeted customer. This might not be a simple task for a vertical market application. One thing we have found that saves us tech support calls is to test on one laser printer and one ink-jet color printer. These two types of printers have different unprintable areas around the margins. It also verifies that we do not have the problem of the hard-coded printer information being stored in the report metadata file (FRX). See the section "How to avoid hard coded printer problems" on page 523 of Hentzenwerke Publishing's *1001 Things You Wanted to Know About Visual FoxPro* for more details on how to deal with this problem.

Make sure graphics are printed both in color and on a black and white laser printer. This will verify that the company logos print correctly and images in the application render clearly on both types of printers. You may want a different logo for color implementations like forms and reports that go to color, and a different image for reports that are output to black and white lasers.

Spell check the report before distributing it with the application. Unfortunately this is a manual process since Visual FoxPro does not have a spell-checking capability (completely removed with Visual FoxPro 7). You could write a tool that checks label objects in the report metadata via Automation with Microsoft Word or one of the third-party spell-checking tools.

Validate that the report has standard items that you require on the report like the date and time the report was run, page numbers, and natural language filtering criteria. The filter criteria are especially important when the user can select different criteria when selecting the data for the report. It is just as handy when the customers call to report a defect that certain data is not showing on the report and you see right away that they filtered it out.

Testing is not complete unless you preview the reports to screen, and output reports to various file formats (ASCII, Excel, HTML, PDF, Word, and so on). This will depend on how

you expose exporting capability for the reporting mechanism. Make sure each export type is tested and that it is opened up in the native editor or viewer. Also make sure when previewing a report that the second and last pages are viewed. It might be obvious why to view the last page since you will want to make sure the entire report is executed and that the final summary information is calculated correctly. Why would the second page be important? It will make sure that the reports run with the `NOWAIT` clause do not have code to close and delete the report cursor.

Verify all formatting is correct. This can be testing the report expression *Format* property, as well as the general layout and format of the report. Make sure that the fonts used on the report are consistent with the other reports throughout the application. We standardized on numeric fields typically aligned on the right, and alphanumeric data on the left. The key is to align all the objects on the report so it looks professional. The alignment tools included in the Visual FoxPro Report Designer are excellent. There is no reason to have objects that are misaligned. Check all label captions for correct spelling.

Test the report with the executable. Many developers ship report metadata files (FRX) separate from the executable so reports can be changed without recompiling the EXE and redistributing it. If this is the case, the EXE needs to be able to find the report on the path, or the path to the reports needs to be included in the `REPORT FORM` code. If the report is included in the EXE, then running the EXE will verify it was included in the project.

Visual FoxPro developers who have used the Report Designer at some point have run up against the "Variable <variable> not found" error message while testing their latest application executable. This message is aggravating since it is displayed without telling you where the expression is flawed in the report. This expression is sometimes difficult to find since there could be dozens of fields on the report. To compound this problem, the expression failure could be in the calculation of fields, calculation of report variables, or Print When conditions. Fortunately, there is a technique that speeds up the tracking of these painful defects. The key to a quick resolution is to suspend the program code after the final report cursors are prepared. If this is not practical, prepare the data manually. Once the data is prepared, modify the report and preview it. The error will be displayed. After you close the report preview mode the Report Designer will display the expression field that the error is occurring. At this point you can make the correction, save the report, and try again.

How can I test business objects?

Business objects are typically invisible objects that enforce business rules for the component. They surely can be tested by interacting with the user interface to make sure they do what they are supposed to do. Since they can be developed independently of the user interface and literally can be developed before the user interface, how can we test them?

Visual FoxPro developers have had the benefit of the Command Window all along. We can instantiate objects and set properties. Results can be dumped to the screen and verified. This works really well. Visual FoxPro 7 also introduced the persistent contents of the Command Window that is an additional benefit for retesting without retyping all the test code. If we want to test out a customer business object's ability to add a record we could enter in all this code.

```
oCust = CREATEOBJECT("cusCustomerBizObj")
oCust.cTableViewList = "v_Customer"
oCust.OpenCursor()

? "Cursor count = " + TRANSFORM(oCust.nCursors)
? "Cursor Alias = " + oCust.cAlias
? "Cursor Reccount = " + TRANSFORM(oCust.nReccount)

oCust.lAllowAdd = tlSubTest

? oCust.New()
oCust.oData.cCustomerID = GUID()
oCust.oData.cCompanyName = "Geeks and Gurus, Inc."
oCust.oData.cWebSite     = "www.geeksandgurus.com"
?oCust.Save()
```

The problem with this technique is that it is only in our Command Window, and occasionally we crash Visual FoxPro and lose the persistent code. Over time we also refine our testing techniques and add more code. The example shown is only testing one of the many features we might have in a business object so even more code is needed to test out the object. So we need a more robust solution than just reentering code in the Command Window.

We have implemented a *SelfTest()* method on each class that supports individual testing. This does not need to be limited to just business objects; it can be implemented for any object. This testing technique encapsulates all the test code for the object so anyone who needs to test it can do so. It allows multiple tests to be written and stored in one central location. There is at least one parameter for the *SelfTest()* method to determine which test we want to run. Optional parameters can also be passed, but keeping these to a minimum will make it easier to execute. Numerous parameters make testing more complicated and difficult to remember. Here is a partial listing of a *SelfTest()* method of the cusBusiness class included in the CH18TESTLANGOPT.VCX class library:

```
LPARAMETERS txTestNumber, tlSubTest

* Code was cut out here that checks the parameter validity
* and translates the character parameter to the numeric test number.

IF VARTYPE(lnTestNumber) # "N"
  RETURN "Bad parameter type passed to SelfTest()"
ENDIF

this.cTableViewList = "v_Customer"
this.OpenCursor()

ACTIVATE WINDOW "Debug Output"
DEBUGOUT "Cursor count = " + TRANSFORM(this.nCursors)
DEBUGOUT "Cursor Alias = " + this.cAlias
DEBUGOUT "Cursor Reccount = " + TRANSFORM(this.nReccount)

DO CASE
  * Add/New
  CASE lnTestNumber = 1
    this.lAllowNew = tlSubTest

    IF this.New()
      WITH this.oData
```

```
        .cCustomerID = GUID()
        .cCompanyName = "Geeks and Gurus, Inc."
        .cWebSite     = "www.geeksandgurus.com"
      ENDWITH

      IF this.Save()
        DEBUGOUT "Added New succeeded"
      ELSE
        DEBUGOUT "Added New - failed save"
      ENDIF
    ELSE
      DEBUGOUT "Added New - failed new"
    ENDIF

  * Delete
  CASE lnTestNumber = 2
    this.lAllowDelete = tlSubTest
    lcCustomer = "IBM"
    this.SetViewParameter("vp_CustName", lcCustomer)
    this.Requery()

    IF this.Locate("cCustomerName = " + lcCustomer)
      IF this.Delete()
        DEBUGOUT "Delete succeeded"
      ELSE
        DEBUGOUT "Delete failed"
      ENDIF
    ELSE
      DEBUGOUT "Delete - unable to locate record"
    ENDIF
ENDCASE
```

So now that we have all the test code encapsulated in the *SelfTest()* method, we can instantiate the object in the Command Window and call the *SelfTest()* method passing the test number parameter. The sample code also demonstrates how you can allow the passing of a character parameter and translate it into a test number.

```
oCust = CREATEOBJECT("cusCustomerBizObj ")
oCust.SelfTest(1)
oCust.SelfTest("New")
```

We recommend testing components extensively. This means that the *SelfTest()* method can get quite lengthy for components with much functionality. The difficult part of generating the *SelfTest()* code is to cover every condition. We have found when writing the *SelfTest()* method that we also refine our requirements and designs. It also gives us a big head-start in developing the test plan for the in-house testing and customer testing.

One question we have been asked is: Do you leave the *SelfTest()* code in the class when deploying? The answer is yes. Some developers use programs to test components. The advantage of this style of testing is that you can easily avoid compiling PRGs into the executable by not including the programs in the project. The advantage of *SelfTest()* methods is that the test code is encapsulated right in the object and you do not need to search for the program that tests the object when changes/updates are completed. If you are concerned with

the additional object code bloating the class, bracket the *SelfTest()* code with an `#IF .T.` and `#ENDIF` when testing and `#IF .F.` and `#ENDIF` before compiling the gold build**.**

How can I test other components?

The previous three sections of this chapter addressed specific components of an application. This section will address some of the remaining components that can comprise the system that is deployed.

The system menu needs to be validated to make sure the functionality designed to expose can be executed. Make sure to select every menu item on every menu pad. Verify that security is implemented correctly by logging in with different user IDs that have different levels of security. Verify that `SkipFor` conditions are all tested and that the correct icons are included on the specified menu items. If you include a "developer only" menu, make sure it is only exposed for the correct user ID, security level, or that it is excluded from the runtime version of the application.

If you are using Visual FoxPro databases and tables be sure to test the reindexing routine (and that the Stonefield Database Toolkit [SDT] metadata was validated if used), and the backup and restore processes function without incident. These features might be included in the executable, or might be tools shipped separately.

ActiveX components are external to the application executable and need to be tested on a machine that is different from the development machines. The same goes for Automation code. Testing these on a separate machine makes sure that the deployment mechanism works, and that the controls and/or applications are installed correctly and registered properly. We have run into issues with automating ActiveX controls that behave differently in production than they did on the development machine. There are licensing issues that can also crop up that can get resolved in testing.

Two other issues that we find developers frequently miss during testing are to check that the system conforms to the Windows color scheme and that the application conforms to the customer's minimum screen resolutions. This is a sensitive issue and a personal one. There are valid reasons that users still use 640x480 screen resolution. Unless you have in the specifications that you can use a higher resolution, be sure to bump down the resolution on your test machines and run through the application user interface. Change the Windows color scheme to a color other than a gray background to see how many of the forms have a hard-coded gray background and how many of the labels and checkbox captions are not transparent.

How do I test systems to verify source code is not in the path?

Always have a separate area on the server/PC for testing the application. This allows you to test the implementation of the system and develop the process to accomplish the implementation. We have a directory for the latest development, a directory for testing, and one more with the latest version of the production system. This way our development staff can separately unit test, while the QA staff can test a release candidate to be sent to the customer once we have completed our in-house testing. If the customer calls in with a support call we can use the production area to see if we can reproduce the alleged defect.

Having the area separate from development will also ensure that the source code is not involved with the testing. Testing the executable as it is shipped will also verify that the source code used to build the executable is not referenced externally unless that is a desired result. Testing in the development environment might hide these errors depending on the path set up for Visual FoxPro.

How do I avoid "Feature not available" errors?

If delivery includes an EXE, always test the EXE standalone. There is nothing worse than having a major feature fail in front of the customer because you forgot to remove a `SUSPEND` from the form you want to show off. Error 1001—"Feature not Available" errors are unacceptable because it shows that the developer never tested the new functionality standalone (see **Table 1**).

***Table 1**. List of commands that trigger "feature not found" errors in a distributed, runtime-based application.*

Unavailable commands	
APPEND PROCEDURES	MODIFY DATABASE
BUILD APP	MODIFY FORM
BUILD EXE	MODIFY MENU
BUILD PROJECT	MODIFY PROCEDURE
CREATE FORM	MODIFY PROJECT
CREATE MENU	MODIFY QUERY
CREATE QUERY	MODIFY SCREEN
CREATE SCREEN	MODIFY STRUCTURE (now works in Visual FoxPro 7 SP1)
CREATE VIEW	MODIFY VIEW
MODIFYCONNECTION	SUSPEND

***Table 2**. List of ignored commands in a distributed, runtime-based application.*

Ignored commands	
SET DEBUG	SET ECHO
SET DEVELOPMENT	SET STEP (new in Visual FoxPro 7)
SET DOHISTORY	

The Microsoft Fox Team added `SET STEP` to the list of ignored commands (see **Table 2**), which was the biggest cause of "Feature not Available" errors, but `SUSPEND` is still trouble. We recommend searching projects for `SUSPEND` commands before compiling the EXE that will be shipped to the customer. The other commands are not commonly used in typical custom application development. The esteemed tech editor of this book, Steve Dingle, has written a Project Search tool that we find extremely handy to find this hidden error. This tool is available at his Web site (www.stevedingle.com).

What are walkthroughs and what are the benefits?

Whil Hentzen refers to the walkthrough process as "Defending Your Life" in his book *The 1999 Developer's Guide*. Another term commonly used to describe this process is a code review. We don't like to call them code reviews since there are so many deliverables in the life

cycle of a project that can be reviewed and have no code. It is a personal, nervous, and beneficial experience. You quickly realize the impact of showing others your developed code. It is a way to test components. Your future code will be better as a result. In a certain sense, a walkthrough is like a trial. Unlike the American judicial system, the developer is guilty until proven innocent by the reviewers.

The basic definition of a walkthrough is simple. You package up all the code you want reviewed and give other developers two or more business days to step through the code looking for errors, defects, performance issues, standards compliance, and support issues. A more complete list is available later in the "Reviewer's responsibilities" section of this chapter. The reviewers tell you what they feel is right and wrong, providing only admiration and constructive criticism. All criticism must be designed to better the code once the suggestions presented are implemented. The review needs to have at least three participants, but four is optimal in our experience. More people can consume too much time and starts to degrade the value of the review. This team is comprised of the original developer and three reviewers. The reason we like three reviewers is that you will always have a majority opinion when there is an issue that is discovered and not agreed upon unanimously as something that needs to be addressed immediately.

The code walkthrough timing is easy to determine. It must be done after the code is completed and before it is released to production. This can be completed during development, after unit testing, during system testing, user acceptance testing, and even beta testing. The later it is done, the less time there is to correct the issues revealed. We recommend walkthroughs be completed after unit testing. Changes made to the code may not be correctly implemented and can break the code that might have worked previously.

One argument we have heard over the years is that walkthroughs delay getting the product to market. This is true in a sense, but in reality it saves time in the overall product lifecycle. Walkthroughs promote better code, which should translate into less production support time. Developers will learn better techniques from each other that make them better and likely faster. Another argument we have listened to is that a walkthrough is not billable time to the client. We also disagree with this statement, but it is up to the individual development shops to make this call. We consider the walkthrough process as an integral part of development, as important as developing and testing the application. (See **Table 3** for a summary of the pros and cons of walkthroughs.) Without development or testing, is the application released? No. The same goes for the walkthrough process.

***Table 3**. Benefits and drawbacks of doing walkthroughs as part of the development lifecycle.*

Benefits	Drawbacks
Readable code	Time consuming
Requirements met	Divisive if not constructive
Syntax defects squashed	Requires mature developers
Standards enforced (and possibly enhanced)	Expense if you decide not to bill hours to clients
Performance addressed	Requires multiple developers
Train development techniques	
Cross training support team	

So what happens if you are a one-person shop? There are three possibilities that we have seen. The first is to partner with another developer, meet for breakfast occasionally, and swap code for review. Each of you will benefit. The second idea is to find the local developer user group and demonstrate pieces of the application, showing off the code. Ask for suggestions and see if anyone points out the problems they see. The last possibility, although not as effective, is to review your own code. We have done this and can say it is difficult at best. If you are going to review your own, get away from the code for a few days or a few weeks before looking it over so that it has time to fade from short-term memory.

What different types of walkthroughs can you do?

There are several different categories of walkthroughs. Each type determines the type of people to invite and the timing of when they need to happen.

Designs/Data models

You can walk through deliverables like system designs, requirement documentation, functional specifications, and data models. These types of deliverables require your more experienced staff. They may even include customers depending on the topic and the sophistication of the customers. Naturally these walkthroughs happen early in the development cycle, even when you are going through multiple iterations of the design.

One-time program/tools

One-time programs may include a quick-and-dirty conversion routine or queries to perform ad hoc reports (that always seem to get to production). Developer tools are usually hacked together quickly to solve some problem or remove some frustration in the daily tasks of development. These rarely need a walkthrough since they are not used by end users. We like to have other developers review some of this code just in case we forget something important. Typically we have one developer quickly check the code and give us the results immediately.

Defect fix

Defect fixes are a common practice in our business. They can range from a simple fix like a syntax error to a complicated logic defect that might lead to a redesign of the feature. The complexity of the fix dictates how many people are involved in the walkthrough. Simple fixes are handled much like one-time programs; complicated changes get the works. Walkthrough timing is important since defect fixes are usually time-critical. The two-business-day rule for review is often waived so the fixes get to the customer quickly. We like to invite some of the rookie developers to learn from the mistakes made in the code in these cases.

Application code

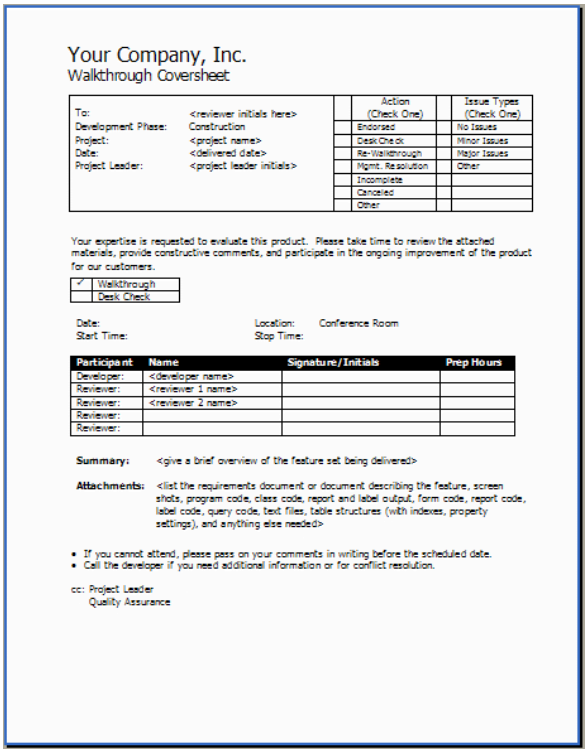
Newly developed features in an application get the works. Enhanced features in an application get the works as well. Is there a difference? In our opinion the only difference is in the amount of code that is reviewed. If the feature is enhanced we only expect the team to spend time on the actual code that has been enhanced. This may be a new method or a new object on a form. Don’t waste precious developer resources walking through code that already has been reviewed. We use developers of all levels for these reviews and they typically happen as soon as the code is unit tested.

Framework/Library code

We know that all the code is important, but the framework code needs extra special attention because it affects every single application that it is based on. There is nothing worse than implementing a change to the report object, putting in a defect, and leaving for the day. It never fails, another developer will be putting the final touches on his release after you leave and every single one of the 200 reports in his application will be broken. Trust us, it happens. Walkthroughs of this magnitude require a special level of developer for the majority of the review team. We still like to involve one rookie programmer so they get some experience looking at code that is "black boxed." This code needs a finer-toothed comb and this requires more time than two business days to review.

How does a developer prepare for a walkthrough? (*Example: WTCover.dot*)

The developer needs to give the reviewers something to review; otherwise, there is no point to this exercise. In our experience we have found that reviewing one feature of an application at a time works best—for instance, a data entry form, a specific report, or a batch process. More than one feature can complicate what the reviewers are evaluating, cause confusion of the requirements, or flat out just take too much time. The material created should take no more than two hours to review and one hour to discuss within a group.



Your Company, Inc.
Walkthrough Coversheet

			Action (Check One)		Issue Types (Check One)
To:	<reviewer initials here>		Endorsed		No Issues
Development Phase:	Construction		Desk Check		Minor Issues
Project:	<project name>		Re-Walkthrough		Major Issues
Date:	<delivered date>		Mgmt. Resolution		Other
Project Leader:	<project leader initials>		Incomplete		
			Canceled		
			Other		

Your expertise is requested to evaluate this product. Please take time to review the attached materials, provide constructive comments, and participate in the ongoing improvement of the product for our customers.

✓	Walkthrough
	Desk Check

Date: Location: Conference Room
Start Time: Stop Time:

Participant	Name	Signature/Initials	Prep Hours
Developer:	<developer name>		
Reviewer:	<reviewer 1 name>		
Reviewer:	<reviewer 2 name>		
Reviewer:			
Reviewer:			

Summary: <give a brief overview of the feature set being delivered>

Attachments: <list the requirements document or document describing the feature, screen shots, program code, class code, report and label output, form code, report code, label code, query code, text files, table structures (with indexes, property settings), and anything else needed>

- If you cannot attend, please pass on your comments in writing before the scheduled date.
- Call the developer if you need additional information or for conflict resolution.

cc: Project Leader
Quality Assurance

***Figure 2**. The cover page for a walkthrough packet informs the developers involved in the review what they are expected to review, who is attending, and when the review is taking place.*

It is most important to have all the code printed out for the walkthrough. A cover page should be attached (see **Figure 2**). This way the reviewers can look at it anywhere, anytime. We prefer to review code at home, relaxing on the couch, and sometimes while watching television. We want to be relaxed. We also treat it as homework and do it at the same time the kids do theirs. All program code should be printed with line numbers so developers have a reference when they get together to discuss the issues found.

*Since much of the source code for Visual FoxPro is stored in DBF metadata files, and the Class Browser is the only way to print out code for classes and forms, we have provided a number of tools in the chapter download to print out source code. These tools are PRINTCX.PRG (forms and classes—see **Figure 3**), PRINTFRX.PRG (reports), and PRINTMNX.PRG (menus). Each of these tools has a report that is used to print the source code. A cover page example is also provided (WTCOVER.DOC).*

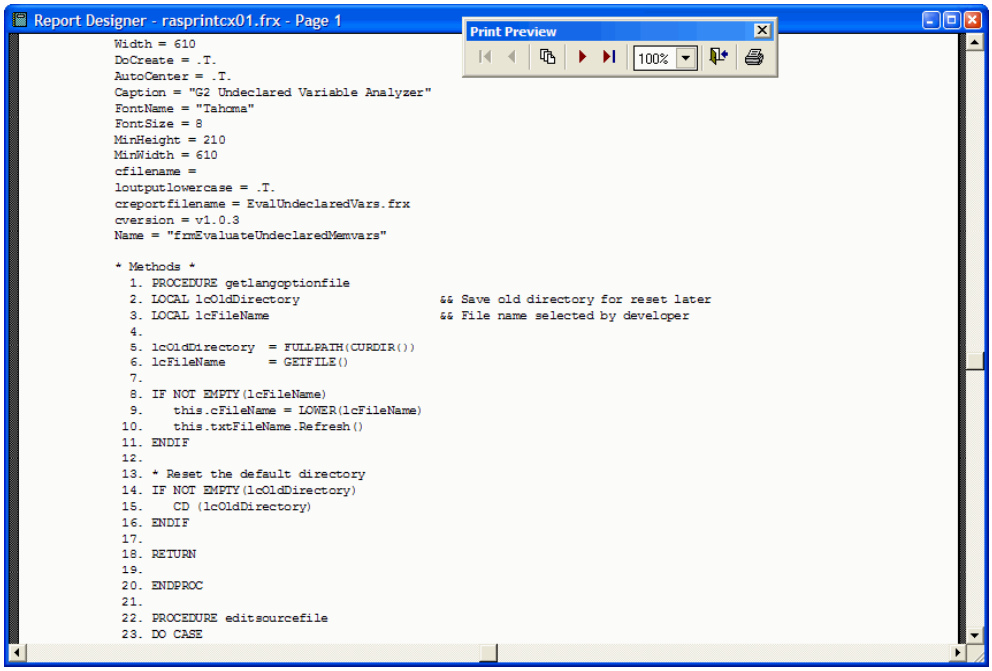


***Figure 3**. The PrintCX utility will print the source code, property settings, and other details stored in the form and class metadata (SCX and VCX).*

The review packet needs to include every bit of code in the feature being reviewed. This includes all code that is developed and called by the feature. Literally include any code that will help the other developers understand whether the feature developed meets the requirements. Here is a list of things recommended for a walkthrough packet. Each walkthrough will likely have a subset of this list.

- Cover page (required)
- Requirements document or section describing feature (required)
- Unit testing results
- Screen shots/Reports/Label output
- Program, Form, Class, Report, Label, Query code
- Text files
- Table structures (with indexes, property settings)
- Anything else needed

In advance the development staff will already be familiar with the company coding standards and guidelines. If an individual is not familiar with them, print out a copy and introduce them to this important documentation. They also need to understand the expectations of the walkthrough process.

There is no need to walk through third-party products that are integrated into the application (although this might be a fun exercise). Also, framework code utilized in the product should not need to be reviewed with each individual feature. Naturally there are exceptions to this rule. One exception might be when the development staff is changing the framework. This would apply to both the enhancements to a third-party framework or any enhancements to your privately developed framework.

The code packets need to be distributed to the reviewers at least two business days in advance of the walkthrough. This gives the reviewers a chance to cover all the material in a reasonable amount of time and allows them to make room in a busy schedule, allowing them to meet their own deadlines.

Picking the review team is important. Each walkthrough might have special requirements. Things we consider when selecting the review team include subject matter expertise, subject matter inexperience (for training purposes), support cross-training, walkthrough experience, workloads, and schedules (like if developers are going on vacation).

We assemble the material and then perform our own walkthrough. It gives us one more opportunity to catch issues before our team sees them. Once we are convinced that the code is ready we reassemble the packet, number the pages to help for reference during the review, and have it copied and distributed to the review team.

What is the reviewer's responsibility of a walkthrough?

The reviewers are like a jury at a trial. They review the facts presented and make a judgment in the end. They need to make sure that the developer upholds the standards and guidelines implemented at the company. They need to check that the requirements are met by the code. Here is a list of items that reviewers need to be sure to look for while reviewing the packet given to them.

- Feature is designed well
- Code meets company standards

- Code is readable
- Code meets requirements
- Test cases available and executed
- Performance issues (optimized code, unnecessary loops, and so on)
- Rushmore optimization (fast data access)
- Appropriate comments, updated comments
- Proper usage of development tool/language
- Proper usage of framework
- Proper scoping of memory variables
- Repetitious code
- Division by zero checks
- Logic structures check other cases (**ELSE**, **OTHERWISE**)
- Return values
- Appropriate data typing
- No hard-coding of paths, file names
- Anything else needed

Each walkthrough will reveal more and more items that you will check and add to the preceding list. It is important to give the review your fullest attention and dedication to finding issues. If there are problems found, give suggestions and possible workarounds. Note articles of your favorite periodical, and point them to other resources like the Help file, or online forums.

What happens during the walkthrough?

So judgment day comes, are you having a Maalox Moment? One thing we constantly need to remind developers is that we do not get together to destroy each other. Well, not the first time walking through the feature. If the code is bad and requires a re-walkthrough and there are major problems the second time around, well, then it is fair game for a tar and feathering.

Start the review on time. This is good advice for any meeting. There are not many ways to waste time that are worse than sitting around waiting for someone to show up late. The developer conducts the meeting. We generally ask if there are any general issues or questions concerning the requirements of the feature. After that we open it up to the reviewers. We ask for the first page that someone has an issue with. We hope no one speaks at this moment, but that never happens. This is why it is important to print out code with line numbers and page numbers in the packet. The reviewers can reference these lines at this time. Start from the first one, move forward to the next one, and continue thorough the packet until the end.

As issues are discussed determine which categories they fall in:

- Must be fixed
- Optionally fixed
- Noted for future standards discussion
- Noted for training topic

These categories are not mutually exclusive. We also refer to the majority wins rule as well. If one developer thinks that one should add 300 lines of comments to describe a `USE` command and the other developers disagree, then we won't make a change. We might note it as a training topic for that developer in the future though. On the other hand, if the developer violated the standard that all commands should be in uppercase and everyone agrees, we will have them fix it before they leave for the day.

We find that these review sessions are an excellent way to introduce the development staff to new techniques, new commands, new design philosophies, and take the time to train them with discussion and heavy use of a whiteboard. As the reviewer, we often have asked the question "Does this work?" Sometimes we do this because we know it does not work, and other times just to get some discussion going on the issue we ask about.

What are the outcomes of a walkthrough?

Once the code has been covered, the time comes to determine the outcome of the "trial." There are several conclusions that can be selected from.

- Endorsed: All things considered, the developer escapes alive with at most minor fixes.
- Desk Check: Issues were discovered and need to have at least one person review fixes.
- Re-walkthrough: Developer loses a pint of blood and has to go through the process again because major issues were revealed.
- Management Resolution: Developer continues to violate good coding practices and has to have corrective action taken by management.
- Incomplete: Walkthrough was interrupted by a natural disaster and nobody was able to reschedule it.
- Canceled: One of the developers sees major issues beforehand and cancels the review so the developer avoids losing a pint of blood or the painful removal of sticky blackened feathers.
- Other: Any other category not covered by the above conclusions.

The conclusion is marked on the cover page and signed-off by the reviewer. The project manager for future reference retains these documents. We like to note some of the issues found to help plan needed training for the development staff. The author has only seen one Management Resolution walkthrough and only a handful of Re-walkthrough conclusions in the past 14 years. This comes from enforcing the "Defending Your Life" mentality. It

usually takes one difficult experience for developers to pick up on what is expected for these review sessions.

What is an alternative to performing a walkthrough?

One alternative to a full walkthrough is a shorter process called the desk check. The desk check is a review that takes place just like a walkthrough, but the review/walkthrough meeting is skipped and the code reviewed is commented/noted on the documents and returned to the original developer for changes. Fewer reviewers are also included in a desk check; typically only one or two are needed. A desk check is needed for quick defect fixes and small changes. It is less formal and quicker, but still important and has the same impact as a regular walkthrough.

Desk checks are also used after a regular walkthrough if enough changes/issues are found and need to be verified that they were implemented as recommended.

Why should I consider hiring someone to test?

Using dedicated testers (also known as quality assurance staff) to make sure your code is working properly can benefit the development team and impress the customers with application deployments that flat out work correctly.

The biggest benefit is that the code gets tested properly by someone other than the developer. Dedicated testing staff will likely be better at testing than the developers since it is their primary responsibility and over time they will develop test patterns (much like we develop and use design patterns) to make sure they thoroughly exercise each part of an application. Also, having a dedicated staff will ensure that the testing gets done before it is released to the customers and frees up the developers to concentrate on developing code that meets the requirements.

Good testing is more complex than sitting someone down at the keyboard and telling them to play with the application. That is only part of the job.

Developers are the worst testers of their own code

Defects, by definition, leak out because programmers did not see the defect in their own code. A lot of times it just takes a second set of eyes to see a defect. This is where professional testers and even other developers can jump in and test the code.

We tended to exercise our code the same way every time. We use our own habits, relying on the mouse a lot, or favoring the keyboard. Dedicated testers will have different habits and different techniques and can quickly uncover a whole slew of defects. They can truly simulate the “stupid user” better than the developer because they have not programmed the code they are testing. When developers watch testers reproduce the defect, they often have one of those whack-the-forehead moments.

Customers are not good at testing applications

Okay, some customers know how to test applications, but it is rare in our experience. They have their own job, and it is not to test software. They also have to make sure they fulfill the responsibilities of their job so that their boss keeps them around. That usually means that testing an application that they will use some time in the future will take a backseat to the real work they have to perform day-to-day.

Besides the fact that the customer has little time to test the application, they should not be testing for the kinds of things that a full- or part-time quality assurance person on your staff is testing. If a customer sees a syntax error when they press the "process payment" button, they are sure to question the level of professionalism you dedicate to your development. The kinds of things customers should be testing for are that the interest rate calculation is generating interest properly, that the reports are formatted in the manner they asked, and that the information on the report is what they need to analyze their business.

The worst thing about this form of testing is the remarkably bad impression you will make of your company. Let us use an example that many of us are familiar with in our own industry, Netscape. They have a reputation for developing a buggy Internet browser. They shipped beta release after beta release with similar results, poor quality. Even though they considered the releases beta, the users only saw problems. By time they got to the production release (which still had numerous defects), the customer base had already lost confidence in the product. This is something you might be able to avoid by having in-house or contracted testers flush out the problems in advance.

You will be a better developer

If you have a part- or full-time tester testing your code and finding defects and you learn from the mistakes you create, chances are your code will get better over time. You will refine the framework to avoid the defects and quirky usability issues. You will begin to recognize the anti-patterns (bad implementations and known good refactoring techniques) as you are coding.

There is a fallacy if testers are hired that the programmers will get sloppy and write buggy code and we can force the programmers to write correct code in the first place if we avoid testers. We feel the opposite is true. Having to expose the developers to rigorous testing by someone else encourages the developers to write better code because their pride is on the line. Our line of thinking only works when developers do take pride in their work, and this is a trait we have ourselves and one we look for in our development staff and contractors.

Avoid the trap that you cannot afford to hire testers

Testing is part of a project and should be considered billable time. It is cheaper for a tester to test than a developer to test because developers typically make more money, plain and simple. We have already stated numerous times that developers make the worst testers, so why not go out and hire testers?

There are a number of sources to find testers. It is our experience that high school and college students looking for experience in the computer industry make excellent candidates for testing positions. It also gives you a chance to see how they work, their dedication to the job, and understand if they might be someone to hire on as a developer after they graduate. They also love a challenge, and what better challenge than to break the senior developer's code?

Finding experienced testers is important as well. They often need to find another challenge once they have "mastered" the current development staff and product line. Retaining good quality assurance people is probably the biggest problem after finding them. This is a tedious job, and it can quickly become boring. Turnover in these positions is not uncommon. The key is for the process to get documented, then to improve the process so the next person coming in gets trained quickly and becomes productive rapidly.

With testers, like programmers, the best ones are an order of magnitude better than the average ones. How can you retain the good ones? Here is a short list of ideas:

- Promote technical support staff to quality assurance or rotate the staff between the two roles if the staff is qualified.
- Share testing resources with other development shops in the area. It keeps the testers fresh working on different projects, and can share some of the salary expense. This works out well if you do not have enough work to keep a person employed full-time on testing.
- Allow testers to develop their careers by taking programming classes, and encourage the better ones to develop automated test suites using programming tools and scripting languages.
- Look for non-traditional computer workers: smart teenagers, college students, and retirees to work part-time. You could create a good testing department with two or three top-notch full-timers and a number of young adults from the local high school working summers to save for college. Check out the local PC user group; they often have retirees as members and you know they already love working on computers.
- Hire temps. If you hire a few temps to come in and turn the crank on your software for a few days, you'll find a tremendous number of defects. Some temps are likely to have good testing skills, in which case it might be worth buying out their contracts to get them full-time. Recognize in advance that some of the temps are likely to be worthless as testers; send them home and move on. That's what temp agencies are for.

Finding good people is hard enough in any business. Recognize that you will have a lot of turnover among your top testers. Hire aggressively to keep a steady inflow of people and make sure to evaluate the work quality of the testing staff. Ultimately, they are going to be responsible for making you shine in front of the customers that purchased the software you are developing.

How can I use the Coverage Profiler to test code?

The Coverage Profiler is a tool provided by Microsoft to analyze executed code for performance (profile mode) and determine what lines of code were executed (coverage mode) during a test run. This tool gives us important information with both sides of the analysis (see **Figure 4**).

The profile mode is an excellent way to determine exactly where the code is slowing down. It shows us how many times each line of code is executed by the number of hits and the length of time the first execution took and how long the average time was for all the executions. In Chapter 12, "Visual FoxPro Tool Extensions and Tips," we demonstrate an add-on to the Coverage Profiler that shows the performance of each line executed. This tool allows us to narrow down the bottlenecks in the code.

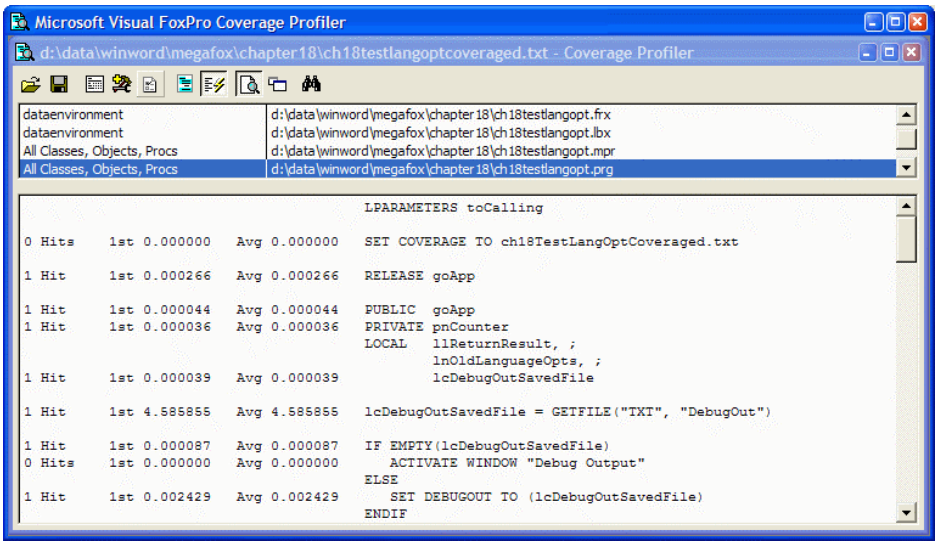


***Figure 4**. The profile mode shows the performance of each line executed.*

The coverage mode is more useful in testing because it helps us ensure that we execute the correct code as we step through our test plan (see **Figure 5**). One tip we like to recommend is changing the character that represents the code that was not executed. We use the question mark (?) because it reminds us to ask the question: Why was this code not executed during this test? You can change this in the Coverage Profiler Options by pressing the fifth button from the left on the toolbar at the top of the main form.

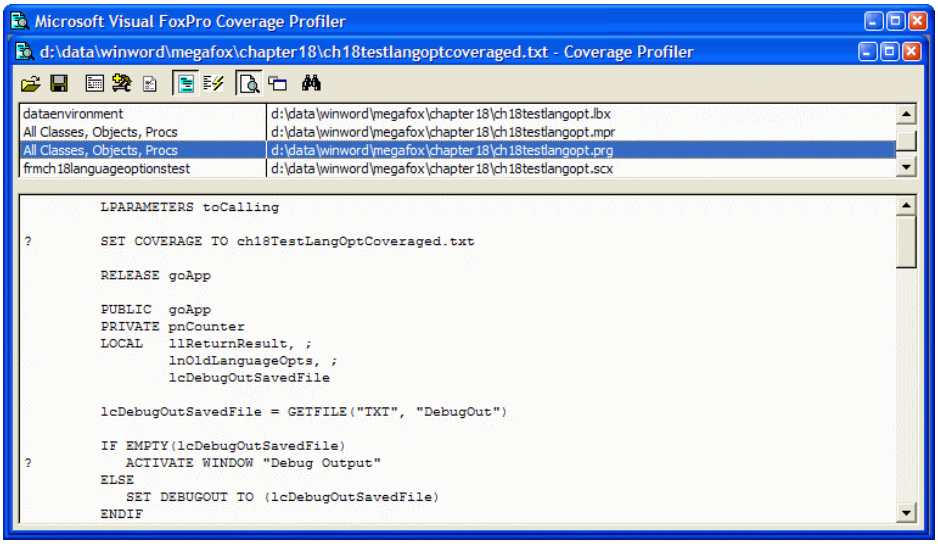


***Figure 5**. The coverage mode shows which lines of code were executed, and more important, which lines of code were not executed during the test.*

What types of automatic test tools are available?

Automated testing tools have been around for years. These tools allow developers to record scripts of mouse movements and keyboard activity as you manually test an application. The script can be run over and over to make sure that the testing is consistent, results are the same, and to eliminate some of the tedious process of testing your applications. There are at least two products available for Visual FoxPro developers. We have not had a lot of experience with either tool, but want to make the information available for our readers to check into if this is something you need.

The first tool that we will introduce is shipped with Visual FoxPro 7, called the Visual FoxPro Active Accessibility Test Harness (see **Figure 6**). This is definitely a 1.0 release, but shows some of the new capabilities available with Visual FoxPro 7 and the ability to leverage the Active Accessibility features for testing purposes. You begin by recording a test session. You need to have your Visual FoxPro application running inside another Visual FoxPro development session (not a runtime EXE). A list of all running Windows applications is presented and you need to select the Visual FoxPro session you want to test. You cannot test non-Visual FoxPro applications. Exercise the application in small increments and save the recorded test scripts. These scripts can be run again and again. You also have the ability to edit the scripts, run pre-script code each time a test is executed, start the Coverage logging when running the scripts, and review the results of the test run.

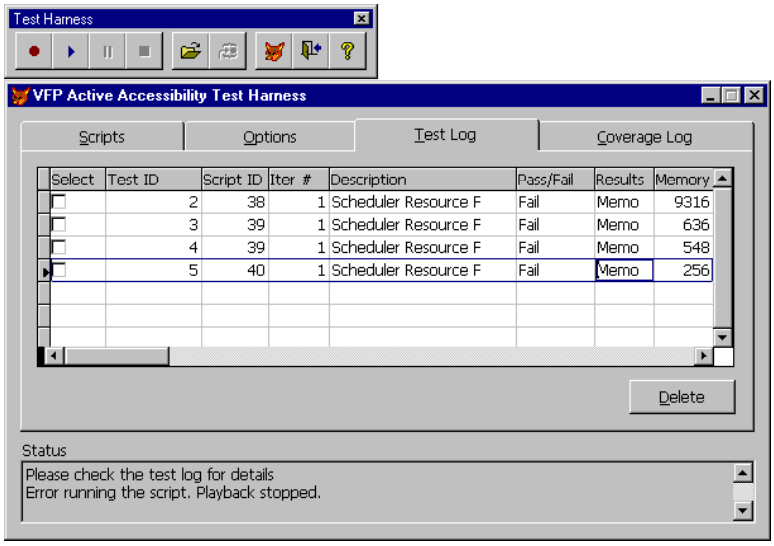


***Figure 6**. The Test Harness provides Visual FoxPro developers with the capability to test their code automatically once a test case is recorded.*

It is definitely not a perfect tool. For one, it does not run under Windows XP, and other OS platforms might have trouble depending on whether the Active Accessibility add-on option is loaded. There are a number of helpful tips and noted limitations documented in the HTML page used for help. Check out the limitation section for tips on working with menus, default command buttons, and combo boxes.

FoxRunner is a commercial package from CAL in Germany (www.cal.de). The marketing literature says it best: "FoxRunner supports software developers in testing their software and in mechanizing frequently repeated tasks." It can be used by developers for testing, sales and marketing for demonstrations, and users for automating repeated tasks. Just like the Visual FoxPro Test Harness, you record scripts, edit them, and play them back when you need to repeat the test. There is where the similarity ends. You can merge scripts, call other scripts from the current script, leverage Cobb Editor Extensions, and analyze property settings as you test the application or component. Interaction with ActiveX controls is also recorded. The professional version also has the ability to generate test data for certain data types (strings, integers, series of digits, numeric values).

COB Editor Extensions (CEE) is a separate free utility for VFP 6.0 and earlier that provides customizable keyword expansion, quick variable scoping, powerful keyboard macros (for indenting, undenting, commenting, and uncommenting blocks of code), and other features. It can be downloaded from www.cobsystem.com.

The Visual FoxPro Test Harness is included with Visual FoxPro, so there is no additional cost. Pricing for FoxRunner starts at US$490 for the Standard version, US$890 for the Professional version, and goes up depending on your licensing needs.

How can I log defect reports?

No matter how much testing is performed, no matter how strong your testing processes are, defects will crop up in all phases of the development cycle. It is a natural part of development. Defects are not a problem if caught early and fixed as soon as practical. The key to dealing with defects is tracking them so that they can be fixed and, just as importantly, logged as being corrected. The benefit to logging both discovered defects and when they are fixed will help better support customers that call up with a problem. Customers love to hear that the defect was previously reported and already fixed.

Developers can only fix defects that they know about. It is important to note that defects can be reported by anyone. It can be developers, quality assurance testers, or customers. Defects can literally be reported by any person who is using the application and perceives a problem. They can also be reported at any point of the development cycle from development though production support.

So what kind of information should be tracked on reported defects?

The following is a suggested list of items that you might want to consider tracking when defects are reported by your development staff, quality assurance testers, or customers. You may find your needs to be different. Application name, application version, who reported the defect, who logged the defect, date/time it was reported, module/component, reproducible steps, result, expected result, any developer notes, test case, how it was reported (phone, e-mail, error log report), priority, severity, OS platform, hardware configuration, regional settings, developer assigned to resolution, status, Internet browser (if Web app), and a tracking number. Once a resolution is found we like to track the resolution (how was the defect fixed),

the version the fix was released, the date it was fixed, who fixed it, the developer who tested the fix, and the customer who tested the fix.

What mechanisms are available to track defects?

There are a number of ways to track defects. The technique used will depend on your needs, how sophisticated the process needs to be, how many people will be reporting defects, and how many people will use the defect tracking process.

You can start out using a paper pad or a word processing document on the network. As defects are reported you add them to the list. If you use a table in the document you can sort items. Most developers find that as additional defects are reported, the paper does not lend to an audit trail because it gets lost or thrown away. The word processing document is definitely a better solution, but can quickly show weakness when it comes to reporting, prioritizing, and searching for solutions.

If you have purchased this book you either are a database developer or have a database developer working for you. The logical next step is to create a database table. This can be maintained with something as simple as a **BROWSE** command. Over time you will refine the table, maybe even create some code tables and relational information to make consistent data entry. Reports can be created and different mechanisms to analyze the information collected over time. You can expect that experience will cause you to recognize what your requirements are, and your data model will get more refined. Sounds like a typical development project. We have found that these situations work their way to building an application. We have also found that building an application in-house is an excellent opportunity for training ourselves or one of our new developers. We have also experienced and heard of similar experiences from our developer friends that defect tracking software typically does most of the job, but that something really important is missing. So developing in-house might be the best fit for the development staff.

So this leads us to the next level, using a third-party defect tracking software package. There are a number of solutions available. One free application is called the Anomaly Tracking System (ATS) that shipped with Visual Studio 97. It was written in Visual FoxPro 5.0 and is available for developers to use. It might be a bit old, but it still provides you with the capability and even a model to work with if you want to develop your own system. Visual Studio 6.0 offered a Web-based version of this application. There are a number of commercial products available as well. Since there are new offerings and evaluations being written, we suggest you go to the Fox Wiki (www.fox.wikis.com) and check out the BugTrackingSoftware topic. Another good site on the Web that provides links to existing tools is www.StickyMinds.com.

There are a couple of Web solutions that are worth checking into. The first one is www.BugCentral.com, which was written in Visual FoxPro and WebConnect. The other site is www.BugHost.com. Both of these services have feature sets that are well thought out. The big benefit of using a Web-based tool is that your customers can directly report defects. They can also query reported defects to see if the problem they are seeing was reported previously and see if it is already fixed. Each of these products has pricing models based on applications and number of users.

How can I test apps on various platforms without reloading the OS?

One of the many challenges that we face as developers is deployment to customers with diverse hardware and operating system configurations. Many of the support calls we get will start with statements like "The application works fine on my co-worker's computer, but not on mine, and I am running Windows 98 and they have Windows XP." These technical support calls can be tough to handle and solve. So how can we better test our applications to ensure they work on various OS platforms? One way is to have a set of test machines with every possible configuration. The other is to get a copy of VMWare Workstation from VMWare.

VMWare Workstation allows you to have virtual machines with different operating systems loaded on one computer without the need to have multiple OS boot options (see **Figure 7**). Once you create a virtual machine you can load one of the many OS platforms (for which you will have to have a license). Configure the virtual machine to reflect the user environment (OS, applications, memory) you need to test. This is really cool since there is virtually no limit to the configurations you can create. You get to specify items like memory, screen resolution, and hard drive space available. If you have users with Windows 98 and Office 97 and a Visual FoxPro 5 app that needs to be supported, you can load all this up and not have to keep a spare old machine hanging around to test things that are reported to the technical support department. Are you having fun supporting IIS 4 on Windows 2000 Server for one customer and IIS 5 on Windows 2000 Server for another? No problem. You can literally test them at the same time on the same PC.

More information and evaluation downloads for VMWare Workstation can be found at www.vmware.com.

Once you have an image of the virtual machine with the OS, you can copy it over and over to load other software that might be involved with different customer configurations. What is really nice is that these virtual machines can be burned to CD or copied to another machine and restored later. This is like having Ghosted images saved up, but does not require the machine to be completely changed each time you want to test on another operating system. Having trouble solving a problem? Copy the virtual machine (it is a file) and send it to a co-worker who also has a license to VMWare Workstation, and have them load it up and track down the problem. No need for them to reformat their computer, just load up the virtual machine.

One other nice feature is that each virtual machine allows the screen resolution to be specified. This allows developers to test applications at various screen resolutions. This is important in the days where you can set the development machine resolution really high and forget that the users might still want to run in 640x480.

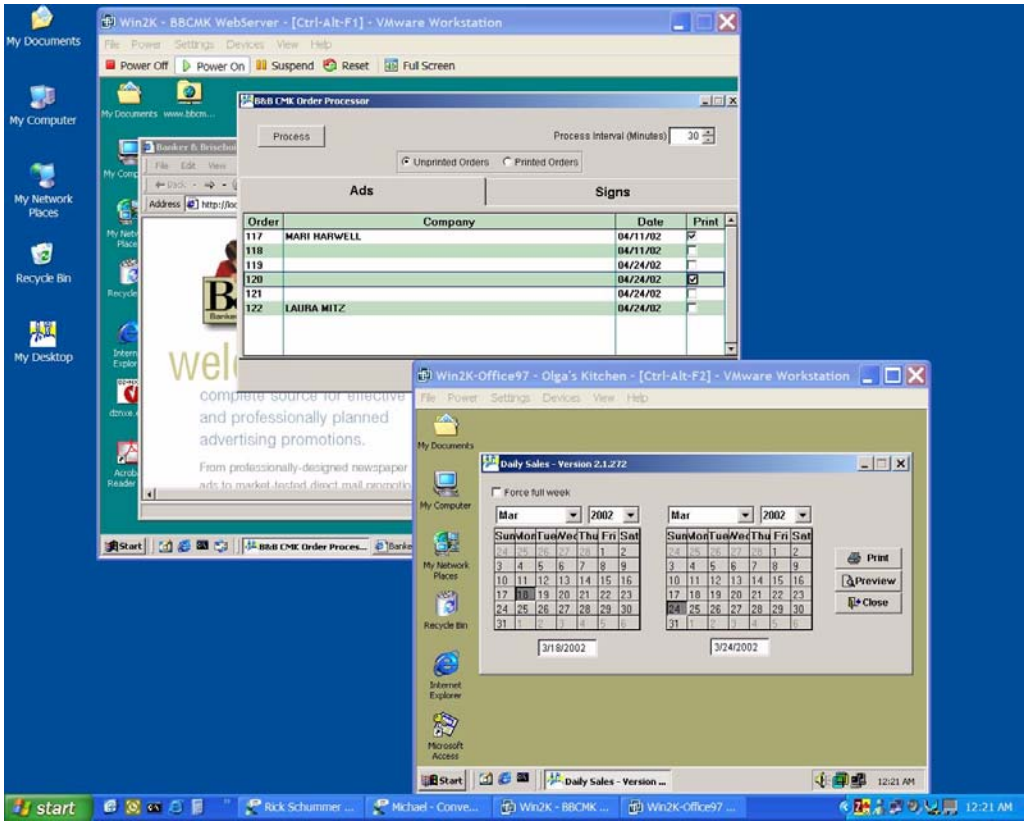


***Figure 7**. This screen shot shows two VMWare Workstation sessions running Windows 2000 within Windows XP Professional testing applications for two different customers.*

This product solves one of the biggest problems facing small shops that do not have the revenue to support a complete test lab. It does require a pretty strong machine (256MB RAM recommended, WinNT4/2000/XP/.NET Server and flavors of Linux as the host OS, 20MB free disk space with 1GB recommended [much more if you have more than a couple virtual machines], Pentium II, III, 4 as well as AMD K6K6-III, Athlon and Duron). These are not unreasonable requirements for today's developers. There is full support for technology like USB, IDE drives, SCSI devices, memory up to 1GB, COM and LPT ports, networking, and sound. The networking is very nice since the virtual machines see each other as well as the host computer.

This technology can also be used outside of the test and support realms. It can be used as a sales tool when you demo your applications in the environment of the potential clients. Training sessions can also benefit from this technology since you can load up a pre-configured environment with the application all set. No need to reset the sample data, change the option settings, load up the latest version and patches, and so forth. Configure it once and load up the virtual machine each time you start a new training session.

Debugging is different from testing

Testing proves that requirements were not met. The next step is to discover why the software was defective. Debugging can be boiled down to two simple words: problem solving. Some developers are good at it, and some developers are not so good at it. What is the difference between the two? Experience and approach.

The debugging process is performed during the construction phase, and after defects are found in unit testing, integration testing, system testing, and user acceptance testing. It is an important part of our jobs as software developers. Defects are discovered and it is our job to find out why this is and determine how it can be corrected.

Experience is an advantage because we can recognize patterns of problems we have solved previously. We learn over time that if a view is not updating correctly that the *SendUpdates* property could be set to false, that a data conflict could be happening during the `TABLEUPDATE()`, that we did not provide data for a required field, that we violated a primary or candidate key with duplication, and so on. Our experience with problems will also dictate what we think are the most common causes for repeat problems. Knowing the common causes of these problems will lead us to solutions faster. Experience will allow us to solve our own problems faster and, maybe even more importantly, be able to solve problems introduced by other developers (from our team, or from a previous developer on a project you joined midstream). The more experience we have developing software will also provide numerous opportunities to solve new problems and see different types of defects, adding to the recognized defect patterns.

There are many approaches to determining the problem and figuring out the solution. The two most common are the shotgun and the scientific method. The shotgun approach is a random way of implementing various code fixes without any rhyme or reason to see if we get lucky and find the solution. We see this approach with less experienced developers. This approach usually takes more time and if a solution is implemented, it is hard to determine what actually resolved the defect and to learn from the original mistake. The scientific approach gives us a basis to observe the problem, formulate questions, predict why it is happening, attempt to fix the problem, and verify it is fixed. This approach is definitely proven to work, requires a formal approach, and produces the additional benefit of learning from the mistakes made. The additional experience also hones our debugging skills so problems can be solved faster the next time the pattern is recognized.

What is the scientific method approach to debugging?

The scientific method is the process by which scientists, collectively and over time, endeavor to construct an accurate (that is, reliable, consistent, and non-arbitrary) representation of the world. This same process can be applied by computer scientists as they endeavor to understand the representation of a requirement and how it is incorrectly constructed in code. The process is broken down into six steps.

Make an observation

The first step to using the scientific method is to have some basis for conducting your research and debugging. This is the step that identifies the problem. The scientific method to debugging and testing software is founded upon direct observation of the code being run. A developer must look critically and attempt to avoid all sources of bias in this observation. But more than

looking, a developer must measure and quantify the observation, which helps in avoiding bias when looking at the problem.

So as we are testing our application (at the sub-component, component, or system level) we recognize and observe a problem or defect. We will use the example of an error "Property <property name> is not found. Unbinding object <object name>" as a form is instantiated. We observe a couple of items. The first observation is the error message that is displayed. The second is that the object is no longer bound to data.

Formulate questions

The second step in the scientific method is to formulate a question. Software developers have to be curious and ask questions! There is one truly foolish question—the one you never ask and never get answered! By asking questions we are elaborating on the problem. At this point we should be asking what happened, what was unexpected, and what did not meet the requirements during the test.

We can ask several questions following our example where the object threw the error message, and did not bind the object. Which object failed to instantiate and threw the error? Is the object bound to a property or cursor column? Did the cursor structure change recently? Did we change any properties on the object that failed to instantiate correctly? Specifically, did we correctly set or incorrectly change the *ControlSource* property of the object that failed?

Create hypothesis/prediction

The next step of our scientific method is to form a hypothesis, and list possible solutions. This is merely an educated guess as to the answer for the question. You gather as much book knowledge and practical experience as you can on the subject to begin to arrive at an answer to your question. This tentative answer, this best educated guess, is our hypothesis.

Please notice that hypotheses do not always have to be correct. In fact, most of science is spent trying to determine the validity of a hypothesis, yet this effort is not likely to give a single perfect answer. So, in formulating your hypothesis, you should not worry too much that you have come up with the best or the only possible hypothesis. The rest of the scientific method will test your hypothesis. What will be important is your decision at the end of the method.

One aspect of your hypothesis is important: It must be able to be rejected. There must be a way to test the possible answer to try to make it fail. If you design an untestable hypothesis, then science cannot be used to help you decide whether it is right or not.

Following our example we can create a hypothesis that states that the *ControlSource* was initially set to a column that existed at the time, but no longer is part of the cursor structure. If the object was bound to a property we can state that the *ControlSource* is misspelled or that the property does not exist. We can predict the reason for each question that we stated in the previous step. You will find that as you gain experience in testing and debugging, you will recognize patterns and find that your hypotheses become more refined.

Fix and test

The prediction is a formal way to put a hypothesis to a test. If you have carefully designed your hypothesis to be sure it can be proven wrong, then you know precisely what to predict. Here you carry out your changes to the code or data structures and compare the results with

the expectations. You need to select one of the hypotheses to test. It is important to construct a test so that it only tests a single hypothesis. Changing more than one constant at a time can make it difficult to prove that the one change was indeed the correct fix to the problem.

Following our example again, based on our experience developing the component, we decide to test the structure change hypothesis. Why did we pick this? Because we overheard at the water cooler 10 minutes earlier that one of our teammates was changing views related to this troubled form. This is where our experience and our observations come into play. We want to test the most obvious and practical solutions first. So we look at the *ControlSource*, and check the view to see if the column exists. We make the appropriate change (either add the column back in, or remove the object from the form). We run the form to see if the results are different or the same.

Evaluate results

How do we evaluate the results? We know what was wrong, and we understand what is expected and correct. As good developers we will try to repeat (replicate) the test several times to avoid the chance of another error occurring. At this point we need to make sure the testing produces the expected results. The testing must meet the requirements. We may also test other test cases involved with the object, component, module, and so on.

Following our example, we want to make sure that we observe that the error message is not displayed and that the object is indeed bound to the correct column of data in the cursor. Further tests should also prove that this data changes, and is saved correctly to prove it is bound properly.

Decision

Now that the test is completed we need to determine whether it was a success or failure. If it was a success, we have finished the test and the scientific method is once again proven reliable and effective. If the test is not successful, we need to loop back to the hypothesis/prediction and pick a new one to fix and test. This iterative process can go on for as many questions as you have posed, predicted why, attempted a fix, and retested. What happens if you run out of questions or possibilities? Naturally you need to come up with more, or ask a colleague to assist you in this process. Watch others use this methodology of testing and debugging to learn from their experience.

In our example we have to decide whether the structure change was indeed the problem. If the error message is still displaying, we have to loop back and see if we correctly changed the *ControlSource* to the proper column, or once again made a typo, or did not add the column back to the view correctly. If it did work we are in good shape and can move on to the next test case or round of testing.

Visual FoxPro debugger tips

The Visual FoxPro debugger is very sophisticated and extremely powerful in helping developers find those menacing defects that creep into our applications. Learning to harness the power of the debugger components can help you figure out the defects faster and assist you in testing the applications to make sure there are no logic defects.

We think it is important to point out the difference between testing and debugging. Testing is the verification that the application meets the requirements. Once the testing reveals

issues, we turn to debugging the error. Debugging is the art of finding the cause of misbehavior and then altering the code so the behavior meets the documented requirements.

Still, debugging is an integral part of the testing process. Therefore, we thought it would be a good place to discuss some additional tips when working with the Visual FoxPro debugger.

How can I set the debugger configuration to factory settings?

(*Example: ClearDebuggerSettings.prg*)

There are many settings and customization capabilities for the debugger in Visual FoxPro for developers to adjust. Every once and a while you might want to just get back to the basics. This can be accomplished with one command:

```
CLEAR DEBUG
```

This command clears all breakpoints, restores the Debugger windows (Trace, Locals, Call Stack, Watch, and Output) to their default positions, clears the expressions in the Watch window, and clears the Output Window. This works in the debugger frame or the FoxPro frame.

There is one reason why you might want to get back to the factory settings. Occasionally we run into C5 errors when using the debugger. Many of the causes have been tracked down and fixed over the years by the Fox Team, but others still linger. A common resolution is to turn off the FoxPro resource file and see if it clears the C5 problem. If the error goes away, it is concluded to be a problem with one or more of the debugger preferences or settings stored in the FoxUser table. One solution typically presented is to delete the FoxUser files and start clean. This is a bit drastic since there are specific records in the FoxUser file that can be removed that provide the same effect. Here is the code that can be run to fix the problem.

```
LOCAL lcOldResource
lcOldResource = SYS(2005)

SET RESOURCE OFF
USE (lcOldResource) EXCLUSIVE ALIAS curResource

DELETE ALL FOR id = "BPOINTS"
DELETE ALL FOR id = "DBGFRAMEM"
DELETE ALL FOR id = "DEBUGFRAME"
DELETE ALL FOR id = "DEBUGGER"
DELETE ALL FOR id = "ETRACK"
DELETE ALL FOR id = "F_DBGWINDOW"
DELETE ALL FOR id = "WATCHEXPR"

PACK

USE IN (SELECT("curResource"))

SET RESOURCE ON
SET RESOURCE TO (lcOldResource)
```

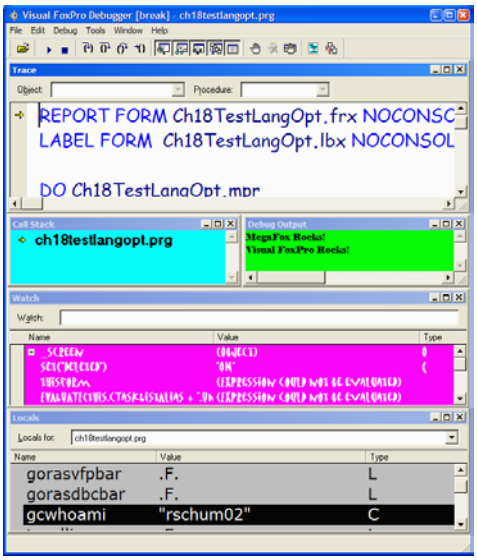


***Figure 8**. The debugger can be configured to select fonts and colors (foreground and background).*

You will find that the color and font settings are not reset after deleting records in the resource file (see **Figure 8**). The reason for this is that the settings are saved in the Windows' Registry under the following key:

```
HKEY_CURRENT_USER\Software\Microsoft\VisualFoxPro\7.0\Options
```

Table 4 lists the Registry values to store the colors for the debugger window.

***Table 4**. Registry values to store the debugger window colors.*

Registry values	
CallstackChangedColor	TraceBreakpointColor
CallstackFontName	TraceCallstackColor
CallstackFontSize	TraceChangedColor
CallstackFontStyle	TraceExecutingColor
CallstackNormalColor	TraceFontName
CallstackSelectedColor	TraceFontSize
LocalsFontName	TraceFontStyle
LocalsFontSize	TraceNormalColor
LocalsFontStyle	TraceSelectedColor
LocalsNormalColor	WatchChangedColor
LocalsSelectedColor	WatchFontName
OutputFontName	WatchFontSize
OutputFontSize	WatchFontStyle
OutputFontStyle	WatchNormalColor
OutputNormalColor	WatchSelectedColor
OutputSelectedColor	

The color settings are not exactly straightforward since they are comma-delimited lists of non-standard RGB() sequences (the first three are the foreground color and the last three are the background color), followed by the determination of the foreground and background colors and whether they are automatic or not automatic (specified by the RGB selection). The font attributes (name, size, style) can be easily handled programmatically via a Registry class like the one shipped with Visual FoxPro as part of the Fox Foundation Classes. The font style is set to zero for normal, one for bold, two for italic, and three for bold and italic.

How can I save and restore the configuration of the debugger?

(*Example: DebuggerConfigSaved.dbg*)

A Visual FoxPro developer can go through a lot of work configuring the debugger with the settings for the watch window, developing the exact breakpoints needed for an application or module, and selecting certain events to be tracked. The settings can change depending on the application or a specific module in an application. We can delete expressions from the watch window and enter in new ones as we test various modules, we can toggle breakpoints in use and not in use, and we can move events that are tracked on and off the list. Another way is to save the exact configuration for the module and later load the configuration without the need to re-enter the expressions or toggle the breakpoints.

This is accomplished via the Debug frame only. Using the menu, you can select File | Save Configuration… to create a file. The file save dialog will default to the current Visual FoxPro directory. To restore a previous configuration you use the File | Load Configuration… menu option. The file contents are stored in an ASCII text file. Here is an example:

```
DBGCFGVERSION=4
WATCH=_screen
WATCH=set("deleted")
WATCH=set("path")
WATCH=thisform
WATCH=curdir()
WATCH=recno()
WATCH=eof()
WATCH=_vfp.ActiveProject
BPMESSAGE=OFF
BREAKPOINT BEGIN
TYPE=2
CLASS=
LINE=0
EXPR=EOF("curReport")
DISABLED=0
EXACT=0
BREAKPOINT END

BREAKPOINT BEGIN
TYPE=3
CLASS=
LINE=0
EXPR="MAIN"$PROGRAM()
DISABLED=1
EXACT=0
BREAKPOINT END
```

```
EVENTWINDOW=ON
EVENTFILE=
EVENTLIST BEGIN
Activate, Deactivate
EVENTLIST END
```

You can manipulate the contents safely and reload the configuration. Make backups of this file if you are worried about breaking the layout.

How can I reorder the contents of the watch window without deleting and re-entering each expression? (*Example: DebuggerConfigSaved.dbg*)

There might be a time when you have numerous watch expressions in the watch window and feel the need to reorder them to a more logical grouping. You could delete and re-enter each expression in the order you prefer, or you can use the following trick to save you a whole bunch of time.

First, save the debugger configuration once you have all the expressions in the watch window that you prefer. Note that you will need to be using the Debugger frame to accomplish this (see the previous section on "How can I save and restore the configuration of the debugger?").

Open up the debugger configuration file with a **`MODIFY FILE`** command. This file is nothing more than a text file. Each of the watch expressions starts with WATCH=, followed by the expression that is evaluated. You can sort the watch items in this file and save it. Load the configuration and the watch window will have the expressions in a new order.

Strangely enough, we decided to try to break this file by sticking WATCH= expressions throughout the file. The expressions showed up in the watch window no matter where we stuck them in the file. We do not support you doing this, just noting the observation.

How can I track which events were triggered in my code?

The current Visual FoxPro debugger has always had event tracking available. This can be set via the Event Tracking dialog in the Visual FoxPro Debugger. Event tracking can also be turned on with **`SET EVENTTRACKING ON`** and off using **`SET EVENTTRACKING OFF`**.

There is a new command in Visual FoxPro 7, **`SYS(2801)`**. This new command lets you decide if you want to only track Visual FoxPro events, Windows mouse and keyboard events, or both. **`SYS(2801,1)`** provides the event tracking that we were used to in previous versions of Visual FoxPro. It provides output like this:

```
56.052, frmevaluateundeclaredmemvars.Activate()
56.122, frmevaluateundeclaredmemvars.MouseMove(1, 0, 587, 152)
58.135, frmevaluateundeclaredmemvars.txtreportfilename.MouseMove(0, 0, 346, 58)
58.165, frmevaluateundeclaredmemvars.txtreportfilename.MouseMove(0, 0, 290, 40)
58.185, frmevaluateundeclaredmemvars.MouseMove(0, 0, 280, 37)
58.215, frmevaluateundeclaredmemvars.txtfilename.MouseMove(0, 0, 248, 26)
58.220, frmevaluateundeclaredmemvars.txtfilename.KeyPress(113, 0)
58.230, frmevaluateundeclaredmemvars.txtfilename.KeyPress(101, 0)
58.225, frmevaluateundeclaredmemvars.txtfilename.MouseMove(0, 0, 228, 21)
58.245, frmevaluateundeclaredmemvars.txtfilename.MouseMove(0, 0, 219, 18)
58.265, frmevaluateundeclaredmemvars.MouseMove(0, 0, 195, 8)
59.417, frmevaluateundeclaredmemvars.Deactivate()
```

If we use the enhanced event tracking provided by **SYS(2801, 2)** we will see only the Windows mouse and keyboard events. Be prepared for an enormous amount of feedback.

```
84.778, MouseMove   00000200 ( 938,  209) 00000000 Visual FoxPro Debugger
84.798, MouseMove   00000200 ( 940,  203) 00000000 Visual FoxPro Debugger
84.858, MouseMove   00000200 ( 590,    0) 00000000 G2 Undeclared Variable
Analyzer
84.888, MouseMove   000000A0 ( 985,  233) 00000014 G2 Undeclared Variable
Analyzer
85.299, MouseMove   00000200 ( 576,    2) 00000000 G2 Undeclared Variable
Analyzer
85.760, MouseUp     00000202 ( 590,   22) 00000000 G2 Undeclared Variable
Analyzerfrmevaluateundeclaredmemvars.txtversion.MouseUp
87.392, KeyPress    00000100 (   1,   32) 68 0
87.532, KeyPress    00000100 (   1,   31) 83 0
88.724, MouseMove   00000200 (   5,  157) 00000000 Command
88.784, MouseMove   00000200 (  10,  257) 00000000
88.784, MouseMove   00000200 (   1,  297) 00000000
88.834, MouseMove   000000A0 ( 280,  582) 0000000A Project Manager - Ch18
88.834, MouseMove   000000A0 ( 280,  591) 0000000A Project Manager - Ch18
88.854, MouseMove   00000200 (  18,  531) 00000000 Microsoft Visual FoxPro
```

SYS(2801, 3) combines both the Visual FoxPro events with the Windows mouse and keyboard events. If you thought you were getting a lot of feedback with them individually, imagine getting both sets together.

The new event tracking provides additional information for the mouse and keyboard events as well. The changes are obvious by evaluating the logs just shown. It should also be noted that the events tracked are within the Visual FoxPro frame (the Visual FoxPro IDE) and the Debugger frame. During our testing we found a gotcha that is important to pass along. If you execute **SET EVENTLIST TO** before executing **SYS(2801)** and turn on event tracking in the debugger's Event Tracking dialog, you will not get any event tracking in the Debug Output window.

How can I track which methods were executed in my code?

The Visual FoxPro debugger has event tracking as we discussed in the previous section, but it does not track method calls natively.

We have had a number of intense discussions with Visual FoxPro developers on the subject of events vs. methods. Events are intrinsic to Visual FoxPro and only provided by Microsoft. We cannot create our own events at this time. Events can be triggered by the user (activating a form, setting focus to a control, clicking on a command button), or they can be triggered programmatically (changing the active page on a pageframe triggers the page activate event, keyboarding a tab key will trigger the *LostFocus* event of one control and the *GotFocus* of another control). There are event methods that are called in response to an event, which are also provided by Microsoft. We can programmatically call our own custom methods from the native event methods.

So how can you easily track the calls through the methods (both native and custom)? Add the following call to each method that you want to track:

```
DEBUGOUT PROGRAM()
```

Our tech editor suggested additional code that provides important information about the calling method or program:

```
DEBUGOUT "Program " + PROGRAM() + ;
         " called by " + PROGRAM(PROGRAM(-1)-1)
```

You will need to make sure the Debug Output window is open when you test out the feature in the application. The contents of the output window can be reviewed directly or saved to a file. If you are doing some regression testing of a component it can be helpful to save the before and after files and then do a comparison of the two to see if anything has changed. You can save the file by right-clicking on the Debug Output window and using the Save As menu option or by using the command **`SET DEBUGOUT TO <filename>.`**

How can I change values of memory variables in the debugger?

Have you ever been debugging some code to find out that a value of a memory variable is not as expected and wished to see how the rest of the code would work if the value was corrected at that point? You can literally change the value of a memory variable as you are stepping through the code in the debugger without the Command Window.

This can be done in the Locals window or the Watch window. Select the memory variable in the Locals window by clicking the mouse on the entry. Click a second time in the Value column, which activates the edit mode for the value. Enter in the value you want the memory variable to be and move off the entry via the Tab key, enter key, or via a mouse click on another item in the debugger.

Now stepping through the code will use the new value and you will be able to evaluate how well the code is working when expected values are used.

How can I ensure variables are declared? (*Example: EvalUndeclaredVars.scx/sct, CH18TestLangOpt.**)

Visual FoxPro 7 introduced a new feature in the debugger that helps track which variables are not declared in our code. This is accomplished by setting the application object's new property called *LanguageOptions* to 1. This setting, combined with the execution of your program code will dump a comma-delimited string of information about undeclared variables to the debugger's output window.

Why is tracking undeclared variables so important to a developer if Visual FoxPro does not require the declaration to run? Visual FoxPro will automatically determine that a new variable is referenced, add it to the list of variables the code is using, and scope it private. Technically it is not important unless the scope of private is a problem in your code. The example we like to use is when a method executes and automatically declares a variable with private scope and calls another method that also has the same variable undeclared (already private in scope). The called method changes the value and returns to the previous method. This side effect could be completely unexpected and lead to a long debugging session because of the confusion.

This new capability works for all code, whether in programs, class methods, form methods, report and label methods (like the dataenvironment), menus, or stored procedures. If it has code, it can be tested. This new capability does have a catch. Visual FoxPro can only

determine undeclared variables in code that it executes. If the code is bracketed by a condition and this condition does not exist during testing, it will not be checked.

We have included a number of sample files named CH18TESTLANGOPT. to test the LanguageOption feature. Start the testing by executing CH18TESTLANGOPT.PRG. You will be prompted to enter in a text file name. If you elect not to pick a file, the Debug Output window will be opened.*

So, to start the logging process we need to execute the following code in a program or from the Command Window:

```
_vfp.LanguageOptions = 1
```

The Debug Output window does not have to be activated to accept the undeclared variable information if you **SET DEBUGOUT TO <filename>**. If the output is not being directed to a file, we suggest activating the Debug Output window so the information can be captured.

```
lcDebugOutSavedFile = GETFILE("TXT", "DebugOut")

IF EMPTY(lcDebugOutSavedFile)
   ACTIVATE WINDOW "Debug Output"
ELSE
   SET DEBUGOUT TO (lcDebugOutSavedFile)
ENDIF
```

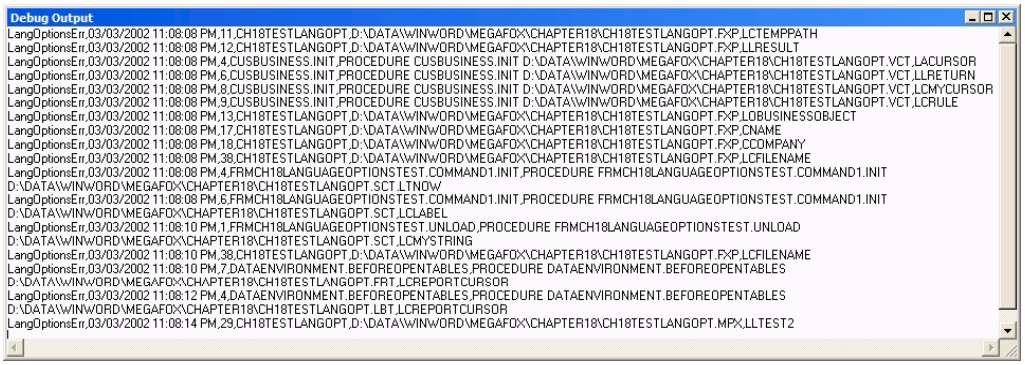


***Figure 9**. Setting LanguageOptions equal to 1 will output a comma-delimited list of details about each variable that is undeclared to the Debug Output window.*

What can we do with a set of comma-delimited strings in the Debug Output window (see **Figure 9**)? We can right-click and use the Save As option to save the contents to a text file. This is the manual way if you did not do a **SET DEBUGOUT TO** at the start of the testing. Once we have the text file we can examine this via a **MODIFY FILE** or **APPEND FORM** into a cursor.

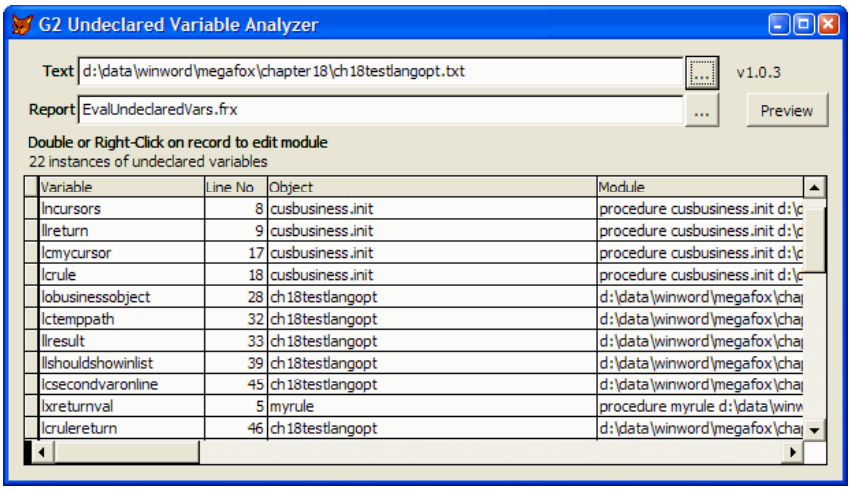


***Figure 10**. The undeclared variable output from the Debug Output window can be saved and analyzed using the EvalUndeclaredVars.scx form.*

One might be asking why someone might go to all the trouble of creating a form (see **Figure 10**) to view the list of undeclared variables. There is one advantage we wanted to exploit, which is that you can double-click or right-click on any entry in the grid and the source code is opened up using the new `EDITSOURCE()` function. This gives us a quicker way to fix all the variable declarations that we feel are necessary.

What are some general tips and tricks for the debugger?

There are a few general tips and tricks when working inside the Visual FoxPro debugger that we find saving us time when testing our code.

Drag and drop

There are several drag and drop opportunities we find ourselves using that we have not seen documented.

You can highlight code in the Trace window and drag it to the Watch window. You can also do the same for variables in the Locals window. The expressions can either be dropped in the textbox or in the evaluated list. If you drop it in the list it is automatically added to the list. If you drop it in the textbox you need to press the Enter key to add it to the list. Naturally you will want to drop expressions that can be evaluated by the Watch window.

You can drag and drop expressions from the Watch window list to the Watch window textbox. This can be useful when you want to add another expression for a different property on the same object or want to add additional levels of containership to the expression.

The contents of the Trace, Watch, Locals, and Debug Output windows can be dragged to the Command Window. This can be handy when testing interactively and you want to jump into the Command Window to do further evaluations of the running code.

Changing expressions

If you have a syntax error or a typo error in the Watch window you can click twice to edit the expression directly in the window. This can be helpful when you entered in a long containership hierarchy and need to correct it quickly, without the need to enter in another entry in the Watch window.

How can I get quick access to the property values of a specific object?

We know this is an old tip from as far back as 1995, but we have had a few new developers cross our paths since then and see some value in repeating the tip once more. The **SYS(1270)** function gets an object reference to the object directly beneath the mouse pointer. We set up a couple of hotkeys to get the object reference:

```
ON KEY LABEL F8 ox = SYS(1270)
ON KEY LABEL F7 RELEASE ox
```

Now you have a reference to look at in the Locals window, **DEBUGOUT**, display in a **WAIT WINDOW**, or even print to the Visual FoxPro desktop. In the debugger you can drill down to look at specific public properties. From the Command Window you can display the property settings as well as call the object's public methods. Make sure to release the object; otherwise, the object will not be able to be destroyed.

Conclusion

We all have horrific stories and experiences that we can tell personally when testing was not done, or not done well. The key to testing is to learn from the mistakes, improve the testing process, and find techniques that lead to defect-free releases. Hopefully, this chapter presented some ideas that will lead you in the right direction.

It is important to remember, while we all strive to make that perfect release and to write defect-free code, that software and the environment that it runs in gets more and more complex. This complexity will likely continue to make our jobs as software developers more difficult. Learning better testing techniques and sharing them with the development community will establish best practices. The better we get as a community, the more our customers will begin to trust in our capabilities as an industry.

Index

Note that you can download the PDF file for this book from www.hentzenwerke.com (see the section "How to download files" at the beginning of this book). The PDF is completely searchable and will provide additional keyword lookup capabilities not practical in an index.

1001 Things You Wanted to Know About Visual FoxPro, 1

Abstract Factory pattern, 518
Abstraction component, 519
Acrobat, 197
Acrobat Forms Author, 220
Active Accessibility Test Harness, 661
Active Messaging, 82
ActiveX Controls, 233
ActiveX Data Controls, 415
ActiveX Data Objects, 614
Adapter Pattern, 538
ADO, 415, 614
ADO Command object, 616
ADO Connection object, 615
ADO RecordSet, 625
ADO RecordSet Object, 619
Amyuni PDF Converter, 199
Animation control, 272
APP extension, 334
ASESSIONS() function, 2
Asynchronous mode, 442
ATAGINFO() function, 3
Attribute, XML, 578
Auto-complete functionality, 49
Automatically update application, 318
Automation, 473
Automation server, 99, 480
AutoRun installations, 358
AUTORUN.INF, 358

Bridge pattern, 518
Browse field names, 6
BROWSE NOCAPTIONS command, 6

Calendar pop up, 9
Canceled printing, 162
Cascading Style Sheet, 543
CDO, 81, 93
CEE, 20, 662
Chain of Responsibility pattern, 518, 526
Change a class name, 395
Charts, 137
Choose printer, 322
Class Browser, 393
Class IS key, 234
Class name, 236
Clean up environment, 1
COB Editor Extensions, 20, 662
Collaboration Data Objects, 81
COM, 471, 580
COM components, error logging, 495
COM DLL, 487
COM Event Binding, 501
COM object constant values, 410
COM servers, 480
COM+, 472
Combo in a grid, 11
Component Object Model, 471, 580
Component Object Model environment, 415
COMRETURNERROR() function, 493
Configure IntelliSense, 53
Connection management, 436
Connection string, 415
Connections, 7
CONNSTRING clause, 429
Constant values, 410
Convert character strings to data, 2
Core data, 26
Coverage log files, 387
Coverage Profiler, 386, 659
Coverage Profiler add-in, 388
CREATEOBJECTEX() function, 475
Creating A COM DLL, 487
CREATOBJECT() function, 474

Crystal Reports, 165
Crystal Reports deployment, 192
Crystal Report Viewer object, 190
CSCRIPT.EXE, 314
CSS, 543
Current record in grid, 16
CURSORTOXML() function, 577
Custom shortcut list, 76

Data driven menus, 29
Data driving techniques, 25
Data Link files, 419
Data migration, 43
Data Source Name, 415
Data validation, 45
Data-driving XML, 584
Date and Time picker, 241
Debugger configuration, 669
Debugging, 629, 666
Decorator Pattern, 535
Delimited list, 5
Design Patterns: Abstract Factory, 518
Design Patterns: Adapter, 538
Design Patterns: Bridge, 518
Design Patterns: Chain of Responsibility, 518, 526
Design Patterns: Decorator, 535
Design Patterns: Elements of Reusable Object-Oriented Software, 517
Design Patterns: Mediator, 530
Design Patterns: Strategy, 522
Design Patterns: Wrapper, 539
Designing COM components, 490
Desktop shortcut, 361
Disable the report toolbar printer button, 160
Disabled menu, 382
Displaying Web content, 108
Distiller, 198
DLL extension, 334, 471
Dmitry Streblechenko, 83
DNS, 82
Document Element, XML, 578
Document Object Model, 114
Domain Name Server, 82
Drill down reports, 185
DSN, 415
DTD, XML, 578
Dynamic Linked Libraries, 471
Dynamic menu captions, 367
Dynamically disable menu bars, 369

Early binding, 475
Edit superclass, 397
EditorOptions property, 53
Element, XML, 578
Email report, 206
Entity Reference, XML, 578
Environment, 1
Erich Gamma, 517
Error 1958, 202
Error loading printer driver, 202
Error logging, COM components, 495
Excel Automation, 148
Exception testing, 634
EXE extension, 334
EXE version numbers, 640
ExecCommand() method, 113
Executable file formats, 334
ExecWeb() method, 112
Export reports to HTML, 184
Export reports to PDF, 184
Export reports to RTF, 184
Export reports to XML, 184
Extended MAPI, 81
EXtensible Markup Language, 577
EXtensible Stylesheet Language Transformations, 606
Extract data, 112

Feature not available error message, 649
Field mapping table, 44
File Server Install, 336
Format text, 35
FoxCode table, 53
FoxRunner, 662
FoxTools Library, 5, 20
FXP extension, 334

G2 Hack menu, 382
G2 Task List Editor, 408
Generalized Markup Language, 577

Generic command buttons, 18
GenMenu.pro, 367
GenMenuX, 371
GETINTERFACE() function, 475
GETWORDCOUNT() function, 5
GETWORDCOUNT() function, 36
GETWORDNUM() function, 5
GETWORDNUM() function, 36
Globally unique ID, 472
GML, 577
GOTO command, 4
Graphic images, 325
Graphs, 137
Grid calendar, 9
GUID, 472

HKEY_CLASSES_ROOT, 305
HKEY_CURRENT_CONFIG, 305
HKEY_CURRENT_USER, 305
HKEY_LOCAL_MACHINE, 305
HKEY_USERS, 305
Hot key, 20
Hyperlink, 117
Hyperlinks in a report, 180

IDispatch, 474
ImageCombo, 259
ImageList, 246
Implementation component, 519
Implementing interface, 497
Importing XML, 591
INI files, 27
INPUTBOX() function, 367
Installation, 335
InstallShield Express, 341
Instancing, 486
Integration testing, 633
IntelliSense, 49
IntelliSense Manager Properties, 67
IntelliSense scripts, 58
Interface, 472
IUnkown, 472

John Vlissides, 517

KiloFox, 1

Late binding, 473
Layered applications, 512
Leftover data sessions, 1
ListView, 249
LOCAL declaration, 20
Local variable declaration, 20
LOCATE function, 4
Lock column in grid, 17

MAPI, 81
Mediator Pattern, 530
Member lists, 61
Menu, 371
Menu bar, 30
Menu templates, 372
Merge module, 345
Merge PDF files, 228
Messaging, 289
Messaging Application Program Interface, 81
Metadata, 26, 367
Monolithic applications, 511
MonthView, 245
Most recently used, 51
Most recently used files, 72
Most recently used list, 61
MPR file structure, 30
MRU, 51
MSChart, 139
MSGraph, 143
MSXML, 580
Multi-threaded components, 481
Multi-threaded DLL, 566
Multimedia MCI control, 272

Named connection, 415
Namespace, XML, 578
Native VFP menu items, 374
New versions of runtime files, 332
Node, XML, 579

Object Browser, 409
Object instantiation, 40
OBJTOCLIENT() function, 9
OCX extension, 235
ODBC, 415

ODBC Manager, 419
Office Web Components, 137, 558
OLE Messaging, 82
OLEDB, 415
OLEPUBLIC, 480
Outlook address book, 100
Outlook object model, 99
Outlook Redemption, 83

Parameterized view, 433
PDF, 197
PDF errors, 202
PDFWriter, 198
Performance with Crystal Reports, 173
Permanently disable menu option, 369
Pop-up menu, 30
Portable Document Format, 197
Preview mode, 157
Print out source code, 653
Print preview, 209
Printing the contents of a Web page, 111
Process data, 26
ProgID, 234
Program header, 71
Project Manager, 412
Projecthook code, 12
Prompt for a printer, 157
PROPER(), 35
Property, 53
Public interface, 472

Query optimization, 8
Quick Info list, 50
Quick info tips, 66

Ralph Johnson, 517
Read mail, 85
Read Outlook mail, 101
Reader, 197
RecordSet Object, ADO, 619
Register a component, 507
Registry class, 306
Registry keys, 312, 348
Regression testing, 637
Regsvr32, 507
Remote view limitations, 433
Remote views, 423
Remove menu pads and bars, 371
Report Designer, 155
Report Designer Component, 189
Reports, 204
Richard Helm, 517
Runtime, 331
Runtime Callable Wrapper, 473
Runtime language resource, 333

SAX, 580
Schema, XML, 579
Self-test() method, 646
Send mail, 90
Send Outlook mail, 104
Service Pack 3, 1
Session base class, 1
Set focus to a control, 13
Setup parameters, 359
Setup Wizard, 341
SGML, 577
SHELLEXECUTE() function, 119, 507
Shortcut menu, 30
Shortcut menu, 375
Simple API for XML, 580
Simple Mail Transfer Protocol, 82
Simple Object Access Protocol, 121
Single threaded components, 480
SKIP FOR clause, 369
SMTP protocol, 81
SOAP, 121
SOAP toolkit, 129
Sound, 274
SPI, 415
SPT, 433
SQL Passthrough, 433
SQL Server 2000 (SPI), 415
SQLCANCEL() function, 436
SQLDISCONNECT() function, 436
SQLEXEC() function, 438
SQLSTRINGCONNECT() function, 416
Standard Generalized Markup Language, 577
Standard graphic images, 328
Status bar, 282

Strategy pattern, 522
Style sheets, 27
Subclass an ActiveX control, 238
Subreports, 187
Superclass, 397
SXDR schema, 598
Synchronous mode, 442
SYS(2004) function, 332
SYS(3054) function, 8
System DSN Registry structure, 420
System testing, 635

Tag existence, 3
Task List, 401
TCP/IP, 83
Test classes, 395
Test plan, 638
Testing, 629
TLB extension, 476
Top-level form menu, 377
Transaction management, 440
TRANSFORM() function, 2
Transmission Control Protocol/Internet Protocol, 83
Transmit error reports, 292
TreeView, 263
Type Libraries, 475

Unattended reports, 204
Unit testing, 632
Updateable cursor, 6, 443
User acceptance testing, 637

Valid XML, 579
Value/Data pairs, 303
Variables list, 74
VBR extension, 476
Version details, 328
VersionIndependentProgID, 234
VFP desktop, 109
VFP7R.DLL, 481
VFP7T.DLL, 481
Virtual Table, 472
Visual FoxPro 6 Setup Wizard, 350
Visual FoxPro Registry settings, 313
VTable, 472

W3C, 544
Walkthroughs, 649
Watch window, 672
Watermarks, 157
Web Browser ActiveX control, 107
Web Service, 566
Web Service Description Language, 121
Web Services, 121
Web Services Meta Language, 569
Well-formed XML, 579
West-Wind Technologies, 81
Windows API, 303
Windows Media Player, 272
Windows print spooler, 155
Windows progress bar, 238
Windows registry, 27, 303
Windows Script Host, 314
Winsock control, 288
WORDNUM() function, 5
WORDS() function, 5
Workstation Install, 336
World Wide Web Consortium, 544
Wrapper Pattern, 539
WSCRIPT.EXE, 314
WSDL file, 121
WSDL Inspector form, 130
WSH, 314
WSML file, 569
Wwipstuff classes, 81
Www.west-wind.com, 81
WZSETUP.INI, 351

XML, 577
XML parsers, 580
XMLDOM, 580
XMLTOCURSOR () function, 577
XPath, 579
XSD schema, 600
XSL, 27, 579
XSL patterns, 607
XSLT, 579, 606